CERN 2000–013
18 December 2000

ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

2000 CERN SCHOOL OF COMPUTING

Marathon, Greece
17–30 September 2000

PROCEEDINGS

Editor: C.E. Vandoni

GENEVA
2000

© Copyright CERN, Genève, 2000

Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document ; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use.

If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.

ISSN 0304–2898
ISBN 92–9083–178–2

CERN 2000-013
18 December 2000

ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

2000 CERN SCHOOL OF COMPUTING

Marathon, Greece
17–30 September 2000

PROCEEDINGS

Editor: C.E. Vandoni

GENEVA
2000

CERN–Service d'information scientifique–RD/1009–1500–décembre 2000

Abstract

The programme of the School was arranged around three themes.

STORAGE AND SOFTWARE SYSTEMS FOR DATA ANALYSIS

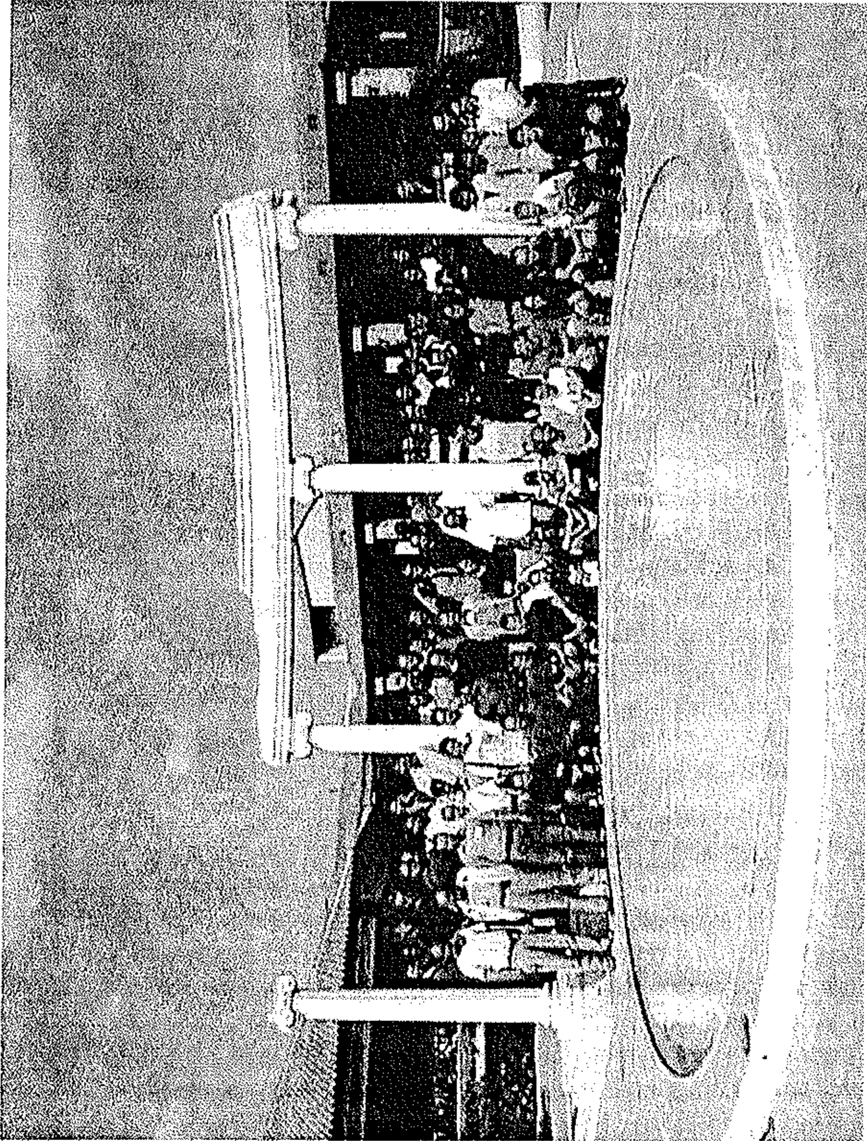
The track addressed the challenge of storing event data and designing analysis systems for future High Energy Physics experiments. The demand for large amounts of reliable and high-availability storage continues to increase more significantly each year. Today everyone can have the same computer, the same hardware, the same network appliances, but no one can have your data. It is the data itself that is the DNA of today's leading-edge organisations. The lectures on storage systems addressed current and long-range directions in data storage highlighting key and emerging technologies, storage architectures, and storage-intensive applications. The lectures on analysis systems provided an overview of the LHC++ and ROOT analysis frameworks, and covered those aspects of software engineering most relevant for HEP software development. The track combined software engineering lectures with exposure to the software technologies and packages relevant for LHC experiments. It showed, in a practical sense, how software engineering can help in the development of HEP applications based on the various data analysis software suites and also gave a taste of working on large software projects that are typical of LHC experiments.

OO DESIGN AND IMPLEMENTATION

Object-oriented concepts are now a fundamental part of the design of many applications. The basic concepts of data encapsulation, polymorphism, and inheritance provide an elegant way to separate the specification of how we interact with a computation from the way that computation is implemented. The course introduced this new paradigm on the design of and the implementation of scientific software packages. The first part of the track focussed on the merit of C++ and Java with respect to the OO paradigm and the second part of the course demonstrated how Geant4 (An Object-Oriented Toolkit for Simulation in High Energy Physics) and JAS (Java Analysis Studio) exploit the advantage of those languages in the context of scientific/engineering computing.

DISTRIBUTED COMPUTING

The requirements of HEP computing for the next generation of experiments at CERN and in the other major HEP laboratories will demand a worldwide access to very large amount of data and massive aggregate computing capacity. High performance networking, WEB access, distributed computing models (such as CORBA) and metaphors such the "GRID" are necessary ingredients of the final solution. This track reviewed the status-of-the-art in high performance networking and introduced concepts such as the GRID, as well as OO methodology such as CORBA.



Foreword

The conviction that experimental physicists and computer scientists have something to learn from each other that was at the origin of the CERN School of Computing in 1970 has never been more true than today. Despite the integration of most of the recent technological breakthroughs and revolutions in computing by the HEP community, we face the challenge of operating world-wide collaborations to prepare unprecedented demands by experiments in terms of data throughput and software engineering. The tools to master data and extract physics to produce publications have to encompass new technologies inherent to the new computing era but are still far out of the reach of most experimental physicists. Distributed systems and object technology are becoming prevalent in response to the rapidly expanding demands of HEP experiments.

The design of distributed systems presents new challenges because it involves the interaction of hardware and software. Continual marketplace innovation drives computing toward heterogeneity and generates a complexity which goes beyond the earlier approaches developed for more homogenous systems executing in a non-distributed environment. This leads to the idea of "middle-ware", i.e. software that presents a uniform interface to applications running on heterogeneous platforms, also providing authentication, security, automatic resource discovery. The evolution of the Internet/Web into an worldwide Information Grid, delivering not only remote data access but also distributed application execution will be made possible only by development of the appropriate "middle-ware".

Yet despite its critical importance, software remains surprisingly fragile; prone to unpredictable performance, dangerously open to malicious attack and vulnerable to failure during implementation. While many engineers and managers are familiar with the basic goals of object technology, only a minority has really understood the deeper concepts and started to apply them thoroughly. Furthermore, despite the most rigorous development processes, software can be assigned tasks beyond its maturity and reliability.

The programme (based on three themes: Distributed Computing, OO Design and Implementation, Storage and Software Systems for Data Analysis) deliberately focuses on how and why these new technologies should be embraced in HEP computing, both from the point of view of world-wide computing resource optimisation and Object Paradigm implementation pertinence. Particular emphasis was put on assessment of the maturing of Software Engineering as a real discipline, trying to evidence the clear gap between vision, education and standard practice. The 2000 CERN School of Computing, organised in collaboration with the Institute of Nuclear Physics NCSR "Demokritos", Athens, provided a challenge to those people who are interested in provocative ideas.

F. Etienne
Chairman of the Advisory Committee

Preface

The 23rd CERN School of Computing took place at the Hotel Golden Cost, Marathon, Greece from 17 to 30 September. The School was organised in collaboration with the Institute of Nuclear Physics NCSR "Demokritos", Athens, Greece.

17 lecturers, of whom six were from CERN, one from Japan, four from the USA and six from Europe, were invited to give courses at the School. One of the six lecturers from CERN was also enrolled as a student. There were also four assistant lecturers. 74 students (coming from 46 institutes, 22 countries and of 24 nationalities attended the School of which 13 were funded by UNESCO; of these 13, 6 also received funding for their travel.

The programme of the Schools was organised round three themes (Storage and Software Systems for Data Analysis, OO Design and Implementation and Distributed Computing) and consisted of 41 hours of lectures (including 3 evening lectures) and 19 hours of exercises. Three evening lectures took place. These were:

M. Griera I Fisa, European Commission, "Application Services Provision"

A. Spyropoulous, "An Introduction to the History of Greek Vineyards"

Y. Maniatis, "High Precision Dating with Carbon-14 as a Tool for Monitoring Cultural and Environmental Events in Prehistory".

The School was opened in the presence of:

Prof. E. Floratos, Greek Delegate to the CERN Council,

Prof. Hans Hoffmann, Director of Technology and Scientific Computing, CERN,

Dr. P. Kokkinias, Institute of Nuclear Physics NCSR "Demokritos",

Mr. C. Vandoni, CERN.

The following members of the Advisory Organising Committee attended the School at various times: R. Jacobsen*, A. Johnson*, W. Carena, F. Etienne (Chairman), F. Flückiger* F. Gagliardi, R. Jones*. G. Kellner, P. McBride.

P. Martucci, IT-PDP group, was the System Manager of the CSC computer centre and his extremely hard work and efficient management, before, during and after the School was a major contributing factor in the success of the School. The computing and peripheral equipment was provided by CERN. Our Greek colleagues provided the network connection from Marathon and handled the details of the Web page. The following Greek assistants were on site, either partially or full-time: Aris Kyriakis, Thanassis Staveris.

R. Marco de Lucas, member of the Local Organising Committee for the 2001 School Committee, attended for two weeks, so as to obtain a better idea of the organisation and setting up of a CERN Computing School.

The local organisation was extremely efficient - our colleagues from NCSR "Demokritos", P. Kokkinias, C. Markou, E. Simoupoulou and C. Zachariadou are to be praised for their help and for their excellent collaboration.

*also lecturer

The hotel Golden Coast was comfortable and well situated and the service was excellent.

Very special thanks must go to the track coordinators and all lecturers for the enormous task of preparing, presenting and writing up their courses.

We express our gratitude to our secretary and administrator, Jacqueline Franco-Turner, not only for the efforts made during the preparation of the School, but also for her invaluable help in preparing these Proceedings.

Finally, we should also thank the students for their enthusiasm and active participation and we wish all of them success in their professional life.

Editor's Note

A number of presentations in electronic form, and, where applicable, some material used for the exercises is available on the web, under

http://www.cern.ch/CSC

A CD-ROM containing a compilation of this material is available on request.

Advisory Committee

W. Carena	CERN, Geneva, Switzerland	
S. Cittolin	CERN, Geneva, Switzerland	
M. Delfino	CERN, Geneva, Switzerland	
F. Etienne	CPPM, Marseille, France	(Chairman)
J. Franco-Turner	CERN, Geneva, Switzerland	(School Administrator)
F. Flückiger	CERN, Geneva, Switzerland	
F. Gagliardi	CERN, Geneva, Switzerland	
L.O. Hertzberger	University of Amsterdam, Amsterdam	
A.J.G. Hey	University of Southampton, Southampton	
R. Jacobsen	University of California, Berkeley	
S. Jarp	CERN, Geneva, Switzerland	
R. Jones	CERN, Geneva, Switzerland	
G. Kahn	INRIA, Le Chesnay, France	
G. Kellner	CERN, Geneva, Switzerland	
A.S. Johnson	SLAC, Stanford, USA	
P. McBride	Fermilab, Batavia, USA	
C. Vandoni	CERN, Geneva, Switzerland	(School Director)

Local Organising Committee

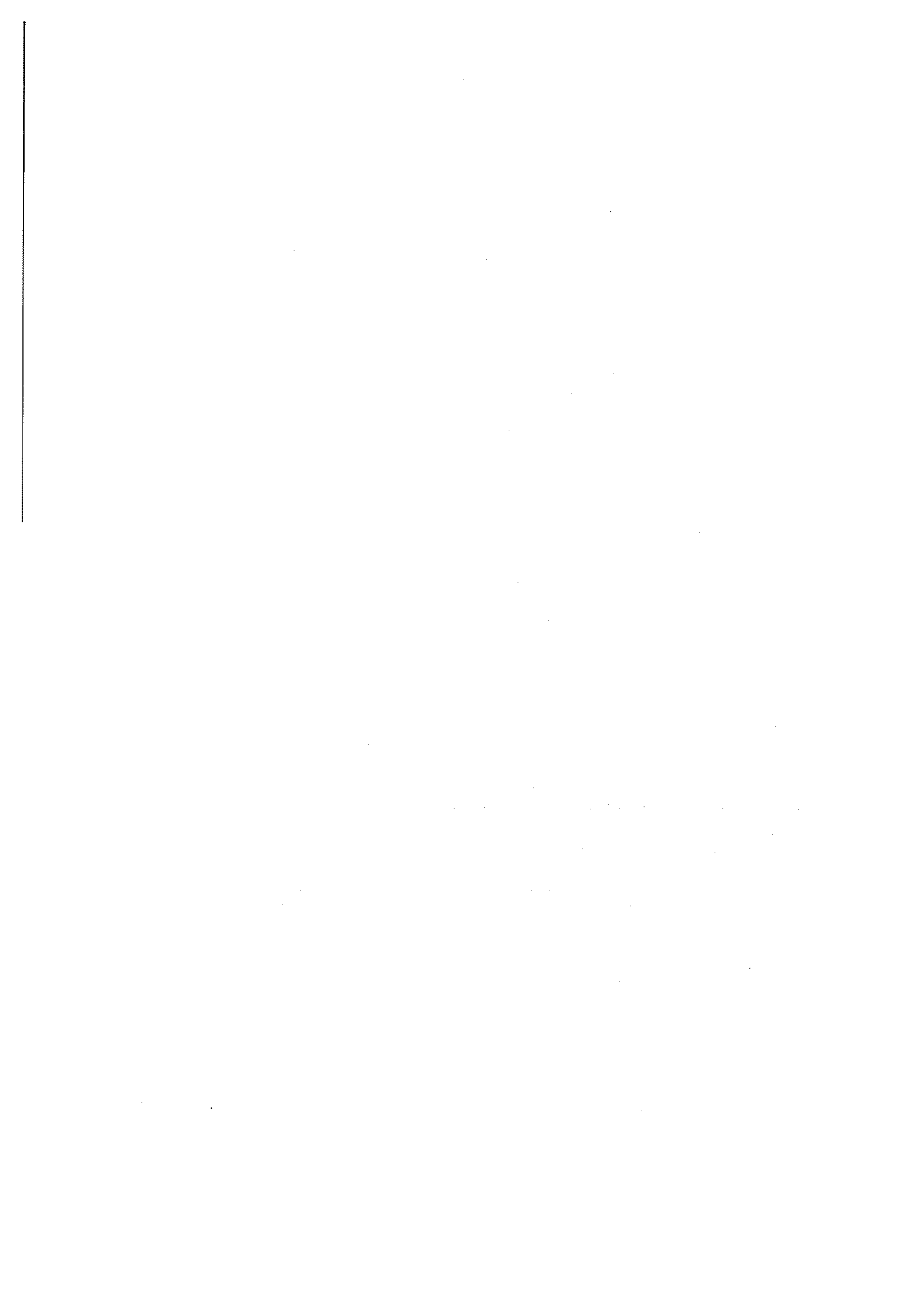
P. Kokkinias	NCSR "Demokritos", Athens, Greece	(Chairman)
C. Markou	NCSR "Demokritos", Athens, Greece	
E. Simoupoulou	NCSR "Demokritos", Athens, Greece	
G. Smyris	Consultant, Athens, Greece	
K. Zachariadou	NCSR "Demokritos", Athens, Greece	

Lecturers

M. Asai	Hiroshima Institute of Technology, Hiroshima, Japan
R. Brun	CERN, Geneva, Switzerland
M. Cafaro	University of Lecce, Lecce, Italy
F. Flückiger	CERN, Geneva, Switzerland
R.G. Jacobsen	University of California, Berkeley, USA
A.S. Johnson	SLAC, Stanford, USA
R. Jones	CERN, Geneva, Switzerland
C. Kesselman	University of Southern California, Marina del Rey, USA
S. Kolos	PNPI, St. Petersburg, Russia
M. Nowak	CERN, Geneva, Switzerland
A. Pfeiffer	CERN, Geneva, Switzerland
F. Rademakers	GSI, Darmstadt, Germany
B.L. Tierney	LBL, Berkeley, USA
J. Moscicki	CERN, Geneva, Switzerland

Assistant Lecturers

G. Cosmo	CERN, Geneva, Switzerland
Z. Molnar	CERN, Geneva, Switzerland
S. Panacek	Fermilab, Batavia, USA
M. Verderi	LPNHE, Palaiseau, France



Contents

Abstract iii
Photograph iv
Foreword v
Preface vii
Advisory Committee ix
Lecturers ix

STORAGE AND SOFTWARE SYSTEMS FOR DATA ANALYSIS

Introduction to Storage and Software Systems for Data Analysis
R.G. Jacobsen 1

Root, An Object Oriented Data Analysis Framework
R. Brun, F. Rademakers, S. Panacek 11

Introduction to The ANAPHE/LHC++ Software Suite
A. Pfeiffer 43

Data Storage and Access in LHC++
M. Nowak 51

Object-Oriented Design of Minimization and Fitting Libraries in the ANAPHE Project
J.T. Moscicki 65

Storage and Software for Data Analysis
R. Brun, R.G. Jacobsen, R. Jones, J. Moscicki, M. Nowak, A. Pfeiffer, F. Rademakers 75

OO DESIGN AND IMPLEMENTATION

OO Design and Implementation
M. Asai 91

OO Design and Implementation: Java and Java Analysis Studio
A.S. Johnson 99

Introduction to GEANT4
M. Asai 107

An Introduction to the Globus Toolkit

G. Aloisio, M. Cafaro 117

DISTRIBUTED COMPUTING

CORBA: a Practical Introduction

S. Kolos 133

A Cache-Based Data Intensive Distributed Computing Architecture for "GRID" Applications

B. Tierney, W. Johnston, J. Lee 155

Improving Quality of Service in the Internet

F. Flückiger 163

List of students 173

INTRODUCTION TO STORAGE AND SOFTWARE SYSTEMS FOR DATA ANALYSIS

Bob Jacobsen
University of California, Berkeley, USA

Abstract
The Storage and Software Systems for Data Analysis track discusses how HEP physics data is taken, processed and analyzed, with emphasis on the problems that data size and CPU needs pose for people trying to do experimental physics. The role of software engineering is discussed in the context of building large, robust systems that must at the same time be accessible to physicists. We include some examples of existing systems for physics analysis, and raise some issues to consider when evaluating them. This lecture is the introduction to those topics.

1. INTRODUCTION

Many modern high-energy physics (HEP) experiments are done by collaborations of hundreds of people. Together, these groups construct and operate complex detectors, recording billions of events and terabytes of data, all toward the goal of "doing physics". In this note, we provide an introduction to how we currently do this, and raise a number of issues to be considered when thinking about the new systems that are now being built.

2. SCALE OF THE EXPERIMENTS

BaBar, CDF and D0 are examples of the large experiments now taking or about to take data in a collider environment. The collaborations that have built these experiments contain 300 to 600 members with varying levels of activity. Almost everybody is considered "familiar" with using computing to do their work, but only a small fraction of the collaboration can be considered as computing professionals. Some of these can even be considered world-class experts in large scale computing. The LHC experiments expect to have collaborations that are a factor of 3 to 6 bigger, with similar distributions of effort.

The scientific method can be summarized as "hypothesis; experiment; conclusion". From proposal to final results of a modern high-energy physics experiment takes several decades. The time-scale of a particular physics measurement is shorter, however. Typically an analysis will take about a year from start to finish, from the idea to measure a quantity to the final result. During this time, the people doing the analysis have to understand their data and the detector's effect on it, connect that to the physical process they want to measure, develop tests of consistency, etc. Much of this work involves the analysis of data from the experiment, and it is here that the computing systems for doing data analysis have direct impact. If the analyst can pose and answer questions quickly, the process of understanding the data and making a measurement will happen faster. If the analysis tools prevent posing sufficiently complex questions, such as detailed examination of the effects of the detector and algorithms, then hard measurements will be made more difficult.

These experiments record large data samples. In the case of BaBar, over a million physics events are recorded each day. All three of the example experiments will be recording hundreds of Terabytes each year for analysis. A large fraction of the collaboration wants to do that analysis, preferably at their home institutes, so it must be possible for a hundred people to simultaneously access the data at locations spread across four continents. Figure 1 shows that even the exponential

increases in cost/performance predicted by Moore's law are not enough to provide the computing power that these growing data samples will need. We'll just have to get smarter!

Modern experiments generally expect to run for almost all of each calendar year. It is no longer possible to process a years worth of data long after it has been taken, allowing time for the alignment and calibration to take place first. Instead, the experiments are moving to "factory" mode of data-taking in which data must be processed and reprocessed on an ongoing production basis.

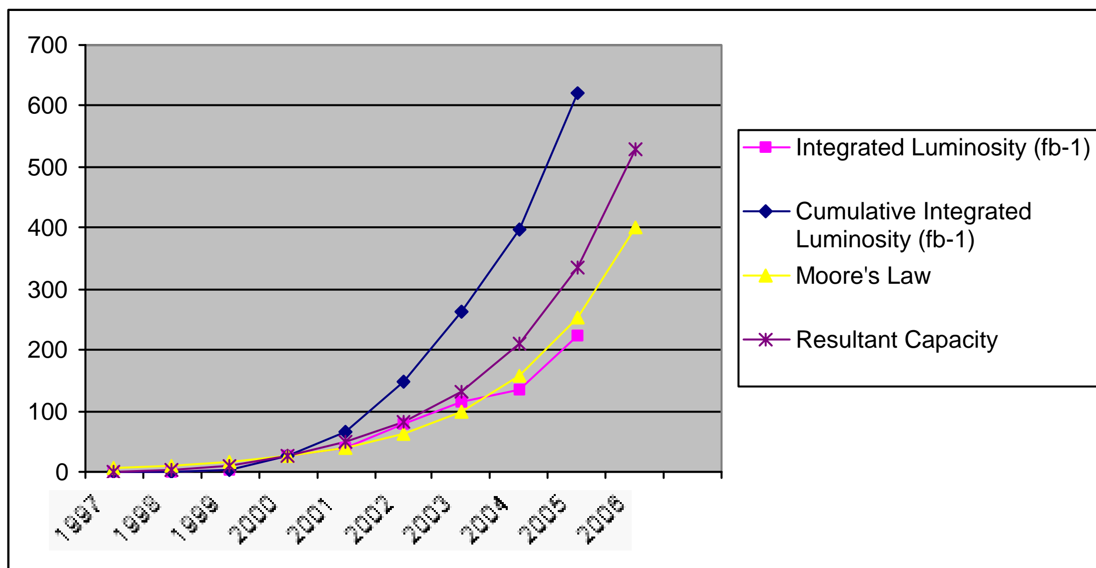


Figure 1 – Moore's law vs. luminosity growth at the BaBar experiment. Figure courtesy R. Mount, SLAC.

3. ANALYSIS MODELS

When large-scale computing was introduced to experimental high-energy physics, operating in "batch mode" was the only available option. This resulted in an analysis model (Figure 2) where the data from the detector was placed in a large tape store, and repeatedly accessed by individual batch jobs. As time went on, we learned how to keep interesting parts of the data on disk storage for faster access, and to use interactive computing systems for developing and testing the programs. This model was used for many years, and produced a lot of important physics results.

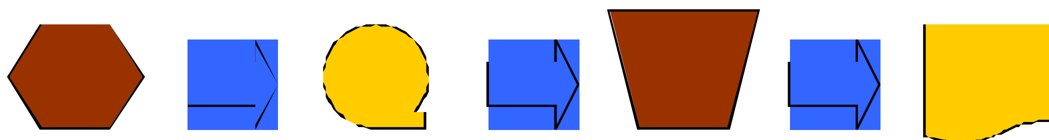


Fig. 2 The batch analysis model. Data moves from the detector (hexagon) to a tape store. Batch jobs are repeatedly run to create the final output of numbers and histograms.

Approximately 15 years ago, the introduction of PAW popularized a new "batch-interactive" model (Figure 3). Large mainframe computers were used in batch mode to efficiently produce summary data in a form that can efficiently be used by smaller workstation computers. This summary data is then interactively processed using a different set of programs to produce the final results.

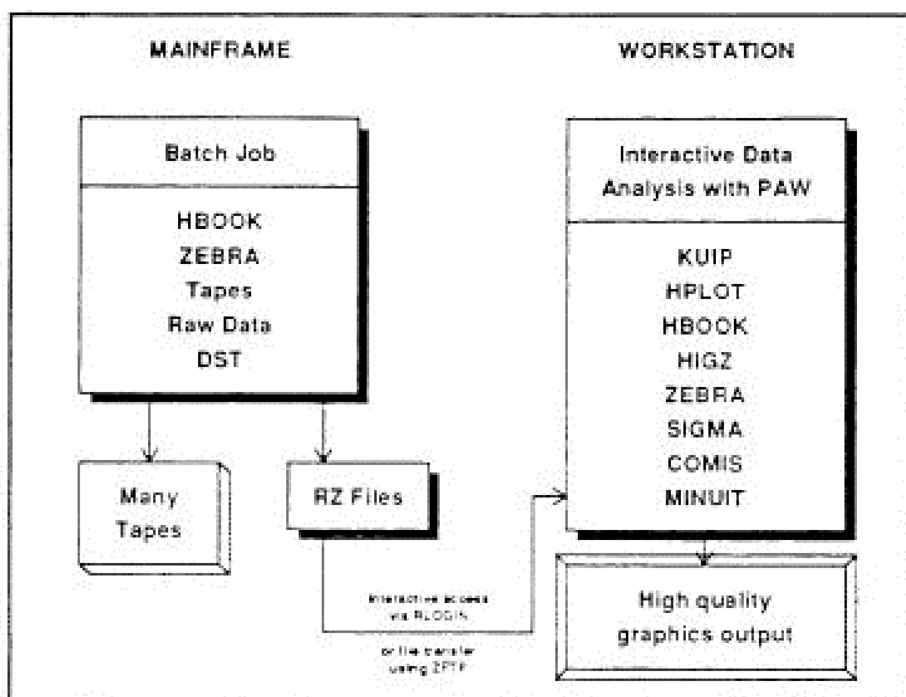


Figure 6.2 Schematic presentation of the various steps in the data analysis chain

Fig. 3 The batch-interactive analysis model. This figure was taken from the PAW manual.

Like many revolutionary advances, this brought both improvements and the seeds of eventual dissatisfaction. The rapid turnaround made possible by interactive workstations made it possible to "PAW through the data", adjusting cuts and improving understanding. This was a qualitative improvement, not just a quantitative one, because the analyst could now make plots to answer many simple questions about as quickly as the plots could be understood. Speeding up the hypothesis-experiment-conclusion loop allowed a more complete understanding of detectors and data than had previously been possible, and allowed more complicated effects to be studied.

Within the past few years, several areas of limitation have been identified with this model. The first of these is the limitations of doing interactive work on individual workstations. We have learned to replace production processing on a single computer with large farms of comparatively cheaper machines. This, combined with the cost/performance benefits described by Moore's law, allowed us to (barely) keep up with the growing size of our data sets, and the increasing complexity of the algorithms we use. Instead, the desktop workstation has become the bottleneck. Second, as more and more complex questions are posed, the need increases to use the same computational tools for interactive analysis as for the large-scale production processing. It is no longer sufficient to look at arrays (ntuples) of numbers with simple cuts; we now want to run large parts of our simulation and reconstruction algorithms as part of our final analysis. This implies that we need to build analysis systems with closer connections between the "batch production" and "interactive analysis" components. In addition, the long time-scale of the LHC experiments has caused concern about the expected lifetime of their software. In the past, typical experiments ran for five or at most ten years. By the end of that time, they had typically outgrown their initial computing systems. Collaborations have historically had great difficulty moving to newer systems while at the same time taking and

analyzing data. The expected twenty year lifetime of the LHC experiments means that the collaborations must have ways to extend and replace their software as needs and technology advance.

As we move toward the larger needs of the LHC experiments, a number of different approaches are being tried to improve the situation; those approaches are what we examined during the Storage and Software Systems for Data Analysis track at the 2000 CERN School of Computing.

4. A SCALING MODEL

The size of a problem affects how best to solve it. For example, teaching one person a particular topic can be done well by having them walk down the hall and ask questions of an expert. This breaks down if sixty people need the knowledge because the expert does not have the time, and typically not the patience, to answer the same questions that many times. A CERN School of Computing involves a lot of logistical and organizational work, but is very well suited to teaching a half-dozen or so topics to about sixty people. For a much larger number, perhaps several thousand, a CERN School is impossible. That many students would want to learn a number of different subjects, at different levels, on different time-scales. Universities have developed to handle that kind of education. We infer from this example that qualitatively different solutions can be needed when the problem size is very different.

We can simplify the issues involved by considering a more abstract case (Figure 4). A particular solution to a problem has a “start-up cost”, a region of application where the effort grows approximately linearly with the size of the problem, and a region where the solution starts to break down resulting in much faster than linear growth.

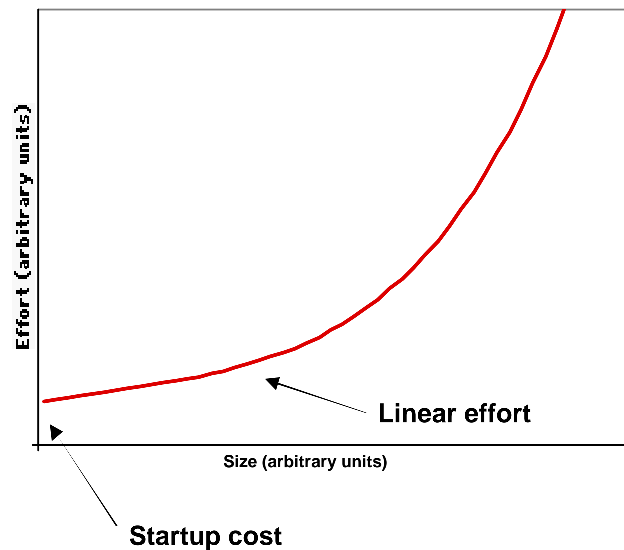


Fig. 4 A model for the cost of solving problems of varying sizes

Now consider what happens when multiple solutions are available for a single problem (Figure 5). For solutions of very small size, clearly the one with the smallest startup cost is the right one. For larger problems, however, you may be better off with a solution that has a flatter curve, i.e. lower slope in the linear region, even if the startup cost is higher.[1] Using a particular solution on problems that are larger than the solution’s linear region is almost always a mistake; the solution will start to break down, and the effort needed to keep it functional will grow without bound.

---

[1] Empirically, we find that almost all solutions that are suited for larger problems have a higher startup cost. Its usually harder to build a system that will scale-up well.

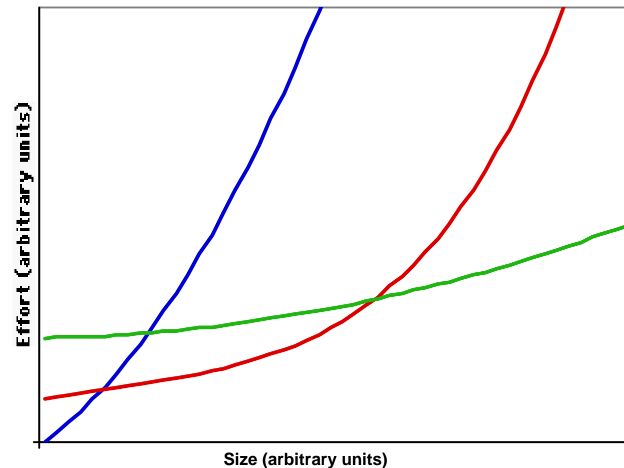


Fig 5 Tradeoffs between different solutions to problems of varying size.

Notice that the size of a problem may change with time. A physicist may write a "quick and dirty" program to solve a specific problem that is not expected to recur. In that case, minimal effort is certainly justified. But that program might become popular, and additional people might want to run it, perhaps on data of slightly different format. This is the range of linear scaling, as people make small modifications to extend the functionality. Eventually, however, these changes can start to conflict; Jane wants to use the program on data without certain characteristics, while John wants to add new functionality that makes heavy use of them. The program becomes more complex, slower, or perhaps less reliable, and eventually can't be extended further. Once that point has been reached, it is necessary to transition to a different solution. This can be costly, and often makes people wish a different solution had been chosen from the beginning.

5. THREE ANALYSIS SYSTEMS

The "Storage and Software Systems for Data Analysis" track at the 2000 CERN School of Computing presented details of two systems currently under development: ROOT and Anaphe/LHC++. In addition, the school also had lectures and exercises on the Java Analysis Studio, JAS. The following sections provide brief summaries of the similarities and differences between these systems. For detail, the reader is referred to the lecture papers later in this report.

One common thread in all of these projects is an effort toward "open development". All three projects make the source-code they develop available to the HEP community. All are interested in getting feedback, especially code changes, from users, and attempt to include them when appropriate. All of the projects restrict write access to their code repository to a team of central developers.

5.1 ROOT

The ROOT project was started in 1995 to provide a PAW replacement in the C++ world. The developers had experience in the creation of CERNLIB, including PAW, and were convinced that the size and lifetime of the LHC experiments required a new system for interactive analysis.

The ROOT developers intend it to be used as a "framework" on which an experimental collaboration will build their offline software.[2] By controlling the link/load process, the event loop, and the class inheritance tree, ROOT provides a large number of powerful capabilities:

---

[2] It is also possible to use the individual ROOT classes and/or libraries separately. For example, experiments at Fermilab are using the ROOT I/O libraries in offline systems without using the entire ROOT framework. Recent work to clarify the structure of ROOT has simplified using it this way.

A sequential object store

Statistical analysis tools for histogramming, fitting and minimization

A graphical user interface, including geometrical visualization tools

An interactive C++ command line

Dynamic linking and loading

A documentation system

A class browser

Numerical utilities

Inter-process communication tools, shared memory support

Runtime object inspection capabilities

ROOT-specific servers for access to remote files

Unique RTTI capabilities

A large set of container classes

These are closely integrated, which makes the power of the system as a whole much larger than can be understood by examining any specific part. Work is actively proceeding to extend the ROOT system in many additional areas, including access to relational and object databases, connections to Java code, classes to encapsulate GEANT4 simulations, parallel execution of ROOT analysis jobs, and others. There are a large number of people actively building on the ROOT framework at several experiments, resulting in a stream of extensions and improvements being contributed back to the ROOT distribution. In effect, the ROOT developers have demonstrated that they can use the large pool of HEP programming talent to build a composite analysis system for the community.

The CINT C++ interpreter allows use of (almost) the same code for both interactive and compiled execution. Users embed their code in the ROOT system by inserting ROOT-specific cpp macros in the C++ definition and declaration files. The ROOT system then uses CINT and other utilities to create schema information and compiled code to perform ROOT I/O, class browsing, etc.

5.2 Anaphe/LHC++

The Anaphe/LHC++ project set out to provide an updated, object-oriented suite of tools for HEP of similar scope to the CERNLIB FORTRAN libraries, with particular emphasis on the long-term needs of the LHC experiment. The strategy is to provide a flexible, interoperable, customizable set of interfaces, libraries and tools that can be populated with existing (public domain or commercial) implementations where possible, and can have HEP-specific implementations created when necessary. Particular attention is paid to the huge data volume expected at LHC, the distributed computing necessary to process and analyze the data, and the need for long-term evolution and maintenance.

Anaphe/LHC++ has defined and is implementing a number of common components for HEP experimental software:

AIDA – abstract interfaces for common physics analysis tools, e.g. histograms

Visualization environment – using the Qt and OpenInventor de-facto standards

Minimizing and fitting – Gemini and HepFitting packages provide implementations, which are being generalized to an abstract interface. Algorithms from both the NAG commercial packages and the CERNLIB MINUIT implementation are included.

HepODBMS – A HEP-specific interface to a general OO event database, used as an object store. The existing implementation uses the Objectivity commercial product.

Qplotter – HEP-specific visualization classes

HTL – Histogram Template Library implementation

LIZARD – an implementation example of an interactive analysis environment

It is anticipated that these components will be used with others, e.g GEANT4, Open Scientist, PAW, ROOT, COLT and JAS, coming from both the HEP and wider scientific communities. Particular attention has been paid to the inter-connections between these, so as to preserve modularity and flexibility. For example, the use of a clean interface and the commonly-available SWIG package makes it possible for an experiment to use any of a number of scripting languages, including TCL, Python, Perl, etc.

Anaphe/LHC++ was aimed at LHC-scale processing from the start. It has therefore adopted tools to ensure data integrity in a large, distributed environment, at some cost in complexity. An example is the use of an object database, including journaling, transaction safety, and location independent storage, for making analysis objects persistent. A physicist writing his or her own standalone analysis program may not see the need for such capabilities, but when a thousand people are trying to access data while its being processed, they are absolutely necessary. Similarly, the project has emphasized the use of modern software engineering practices such as UML, use cases, CASE tools, etc, to improve the quality and long-term maintainability of the code.

Anaphe/LHC++ should not be considered as a unique base upon which an experiment builds a monolithic software system by creating concrete classes. Rather, by identifying interfaces for components likely to be present in analysis software, Anaphe/LHC++ intends to provide a basic structure that can grow and evolve over the long term, perhaps even as HEP transitions from C++ to Java to TNGT (The Next Great Thing).

5.3 Java Analysis Studio

The strategy of the Java Analysis Studio developers is to leverage the power of Java as much as possible because

It provides many of the facilities needed as standard

Its capabilities are advancing fast

It is easy to learn and well-matched in complexity to physics analysis

It is a mainstream language, so time learning it is well spent

It is a productive language, e.g. no time wasted on core dumps

JAS's focus is primarily on the computational part of the analysis task. As such, it uses defined interfaces, called "DIMs", to attach to an experiment's own method of storing and retrieving data for analysis. JAS is not intended as the basis for creating the production simulation and reconstruction code for an experiment. Rather, JAS interfaces exist or are being defined to attach JAS to other parts of common HEP code, including the GEANT4 simulation package, AIDA for histogramming, WIRED for event display, StdHEP for Monte Carlo simulated events, and similar experiment-specific code. The simple "plugin" architecture is intended to make it convenient to add interfaces by adding C++ code to an existing system. Direct connections to existing C++ code, without creating an explicit interface, is currently a weak point of Java.

Java itself provides many of the desired tools, such as class browsers, object inspection tools, documentation systems, GUI and visualization classes, collection classes, object I/O, inter-process communication, etc. This allows JAS to benefit from the efforts of the world-wide community of

Java tool developers, a much larger group than just HEP programmers. As an example, there are ongoing Java efforts to create GRID-aware tools for distributed computation which can be interfaced for use by JAS. It is expected that large-scale tests of distributed analysis using these tools can be done in the next year.

6. SUMMARY AND CONCLUSIONS

The LHC experiments present us with a dilemma. They will produce large amounts of data, which must be processed, analyzed and understood. Current experiments are now solving problems about order of magnitude smaller, but only by working at the limits of the capability of available people and technology. The several analysis systems now under development promise to improve our capabilities, perhaps even change the way we work. All of these systems have proponents and detractors, strengths and weaknesses. They have taken very different approaches to solving the same basic problems. Over the next years, as the LHC experiments develop and deploy their choices for production and analysis systems, the community needs to profit from the best qualities of each of these systems.

ACKNOWLEDGEMENTS

We wish to thank the organizers of the 2000 CERN School of Computing for a stimulating school. We also thank the developers of ROOT, Anaphe/LHC++ and JAS for both their help with the school and their work to provide to tools to the community.

BIBLIOGRAPHY

The Mythical Man-Month; Fred Brooks; Addison Wesley

Peopleware: Productive Projects and Teams; Tom Demarco and Timothy Lister; Dorset House

The Deadline: A Novel about Project Management; Tom Demarco; Dorset House

The Cathedral and the Bazaar; Eric Raymond; O'Reilly

Designing Object-Oriented C++ Applications Using the Booch Method; Robert Martin; Prentice Hall

Large Scale C++ Software Design; John Lakos; Addison-Wesley

The Computing in High Energy Physics (CHEP) conference series is a good source of information on current activities in this area. The most recent one was this past February in Padova Italy. The conference proceedings have been published, and are on the web. Some of the most relevant papers are listed below.

A108 - The Design, Implementation and Deployment of a Functional Prototype OO Reconstruction Software for CMS. The ORCA Project. D. Stickland et al.

A152 - GAUDI - The Software Architecture and Framework for Building LHCb Data Processing Applications. LHCb Computing Group

A245 - The Physical Design of the CDF Simulation and Reconstruction Software. Elizabeth Sexton-Kennedy et al.

A264 - Rapid Software Development for CLEO III. M. Lohner et al.

A326 - The STAR Offline Framework. V. Fine et al.

C103 - Operational Experience with the BaBar Database - Q. Quarrie et al.

C201 - The CDF RunII Event Data Model. R. Kennedy et al.

C240 - Event Data Storage and Management in STAR. V. Perevoztchov

C367 - The CDF RunII Data Catalog and Data Access Modules. P. Calafiura et al.

E248 - Software Sharing at Fermilab. R. Pordes et al.

F033 - CMT: a software configuration management tool. C. Arnault

F175 - ROOT for Run II. P. Canal et al.

F202 - SoftRelTools rev 2 at Fermilab. J. Amundsen et al

ROOT, AN OBJECT ORIENTED DATA ANALYSIS FRAMEWORK

René Brun, Fons Rademakers, Suzanne Panacek
CERN, Geneva, Switzerland

Abstract
ROOT is an *object-oriented framework* aimed at solving the data analysis challenges of high-energy physics. Here we discuss the main components of the framework. We begin with an overview describing the framework's organization, the interpreter CINT, its automatic interface to the compiler and linker ACLiC, and an example of a first interactive session. The subsequent sections cover histogramming and fitting. Then, ROOT's solution to storing and retrieving HEP data, building and managing of ROOT files, and designing ROOT trees. Followed by a description of the collection classes, the GUI classes, how to add your own classes to ROOT, and PROOF, ROOT's parallel processing facility.

1 INTRODUCTION

In the mid 1990's, the designers of ROOT had many years of experience developing interactive data analysis tools and simulation packages. They had lead successful projects such as PAW, PIAF, and GEANT, and they knew the twenty-year-old FORTRAN libraries had reached their limits. Although still very popular, these tools could not scale up to the challenges offered by the Large Hadron Collider, where the data is a few orders of magnitude larger. At the same time, computer science had made leaps of progress especially in the area of Object Oriented Design, and the time had come to take advantage of it.

The first version or ROOT, version 0.5, was released in 1995, and version 1.0 was released in 1997. Since then it has been released early and frequently to expose it to thousands of eager users to pound on, report bugs, and contribute possible fixes. More users find more bugs, because more users add different ways of stressing the program. By now, after six years, many users have stressed ROOT in many ways, and it is quiet mature.

ROOT is an *object-oriented framework*. A *framework* is a collection of cooperating classes that make up a reusable solution for a given problem. The two main differences between frameworks and class libraries are:

Behavior versus Protocol: A class library is a collection of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the rules for behaviors that can be combined.

Implementation versus Design: With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related classes work together.

Object-Oriented Programming offers considerable benefits compared to Procedure-Oriented Programming:

Encapsulation enforces data abstraction and increases opportunity for reuse.

Sub classing and inheritance make it possible to extend and modify objects.

Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.

Complexity is reduced because there is little growth of the global state, the state is contained within each object, rather than scattered through the program in the form of global variables.

Objects may come and go, but the basic structure of the program remains relatively static, increases opportunity for reuse of design.

1.1 Main Components of ROOT

- A hierarchical object-oriented database (machine independent, highly compressed, supporting schema evolution and object versioning)
- A C++ interpreter
- Advanced statistical analysis tools (classes for multi-dimensional histogramming, fitting and minimization)
- Visualization tools (classes for 2D and 3D graphics including an OpenGL interface)
- Advanced query mechanisms to select information in very large data sets (ROOT Trees)
- A rich set of container classes that are fully I/O aware (list, sorted list, map, btree, hashtable, object array, etc.)
- An extensive set of GUI classes (windows, buttons, combo-box, tabs, menus, item lists, icon box, tool bar, status bar and many more)
- An automatic HTML documentation generation facility
- Run-time object inspection capabilities
- Client/server networking classes
- Shared memory support
- Multi-threading support
- Remote database access either via a special daemon or via the Apache web server
- Ported to all known Unix and Linux systems and also to Windows 95 and NT

1.2 The Organization of the ROOT Framework

The ROOT framework has about 460 classes grouped by functionality into shared libraries. The libraries are designed and organized to minimize dependencies, such that you can include just enough code for the task at hand rather than having to include all libraries or one monolithic chunk.

The core library (`libCore.so`) contains the essentials; it needs to be included for all ROOT applications. `libCore` is made up of Base classes, Container classes, Meta information classes (for RTTI), Networking classes, Operating system specific classes, and the ZIP algorithm used for compression of the ROOT files.

The CINT library (`libCint.so`) is also needed in all ROOT applications, but `libCint` can be used independently of `libCore`, in case one only needs the C++ interpreter and not ROOT.

Figure 1shows the libraries and their dependencies. For example, a batch program, one that does not have a graphic display, which creates, fills, and saves histograms and trees, only needs the core (`libCore` and `libCint`), `libHist` and `libTree`. If other libraries are needed ROOT loads them dynamically. For example if the `TreeViewer` is used, `libTreePlayer` and all the libraries the `TreePlayer` box below has an arrow to, are loaded also. In this case: `GPad`, `Graf3d`, `Graf`, `HistPainter`, `Hist`, and `Tree`. The difference between `libHist` and `libHistPainter` is that the former needs to be explicitly linked and the latter will be loaded automatically at runtime when needed. In the diagram, the dark boxes outside of the core are automatically loaded libraries, and the light colored ones are not automatic. Of course, if one wants to access an automatic library directly, it has to be explicitly linked also.

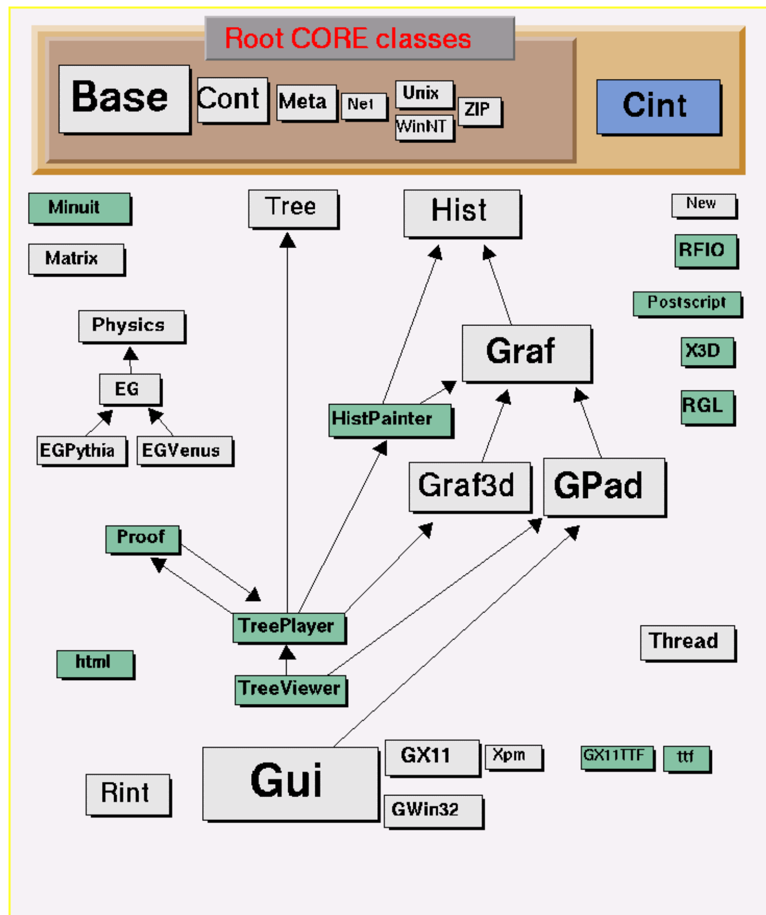


Figure 1: The ROOT libraries and their dependencies

Here is a short description for each library, the ones marked with a * are only installed when the options specified them.

- libCint.so is the C++ interpreter (CINT).
- libCore.so is the Base classes.
- libEG.so is the abstract event generator interface classes.
- *libEGPythia.so io the Pythia5 event generator interface.
- *libEGPythia6.so is the Pythia6 event generator interface.
- libEGVenus.so is the Venus event generator interface.
- libGpad.so is the pad and canvas classes which depend on low level graphics.
- libGraf.so is the 2D graphics primitives (can be used independent of libGpad.so).
- libGraf3d.so is the3D graphics primitives.
- libGui.so is the GUI classes (depends on low level graphics).
- libGX11.so is the low level graphics interface to the X11 system.
- *libGX11TTF.so is an add on library to libGX11.so providing TrueType fonts.
- libHist.so is the histogram classes.
- libHistPainter.so is the histogram painting classes.
- libHtml.so is the HTML documentation generation system.

- libMatrix.so is the matrix and vector manipulation.
- libMinuit.so - The MINUIT fitter.
- libNew.so is the special global new/delete, provides extra memory checking and interface for shared memory (optional).
- libPhysics.so is the physics quantity manipulation classes (TLorentzVector, etc.).
- libPostscript.so is the PostScript interface.
- libProof.so is the parallel ROOT Facility classes.
- *libRFIO.so is the interface to CERN RFIO remote I/O system.
- *libRGL.so is the interface to OpenGL.
- libRint.so is the interactive interface to ROOT (provides command prompt).
- *libThread.so is the Thread classes.
- libTree.so is the TTree object container system.
- libTreePlayer.so is the TTree drawing classes.
- libTreeViewer.so is the graphical TTree query interface.
- libX3d.so is the X3D system used for fast 3D display.

1.3 CINT: The C/C++ Interpreter

A key component of the ROOT framework is the CINT C/C++ interpreter. CINT, written by Masaharu Goto of Hewlett Packard Japan, covers 95% of ANSI C and about 85% of C++ (template support is being worked on, exceptions are still missing). CINT is complete enough to be able to interpret its own 70,000 lines of C and to let the interpreted interpreter interpret a small program.

The advantage of a C/C++ interpreter is that it allows for fast prototyping since it eliminates the typical time-consuming edit/compile/link cycle. Once a script or program is finished, you can compile it with a standard C/C++ compiler (gcc) to machine code and enjoy full machine performance. Since CINT is very efficient (for example, for/while loops are byte-code compiled on the fly), it is quite possible to run small programs in the interpreter. In most cases, CINT out performs other interpreters like Perl and Python.

Existing C and C++ libraries can easily be interfaced to the interpreter. This is done by generating a dictionary from the function and class definitions. The dictionary provides CINT with all necessary information to be able to call functions, to create objects and to call member functions. A dictionary is easily generated by the program `rootcint` that uses as input the library header files and produces as output a C++ file containing the dictionary. You compile the dictionary and link it with the library code into a single shared library. At run time, you dynamically link the shared library, and then you can call the library code via the interpreter. This can be a very convenient way to quickly test some specific library functions. Instead of having to write a small test program, you just call the functions directly from the interpreter prompt.

The CINT interpreter is fully embedded in the ROOT system. It allows the ROOT command line, scripting and programming languages to be identical. The embedded interpreter dictionaries provide the necessary information to automatically create GUI elements like context pop-up menus unique for each class and for the generation of fully hyperized HTML class documentation. Further, the dictionary information provides complete run-time type information (RTTI) and run-time object introspection capabilities.

On the ROOT command line, you can enter C++ statements for CINT to interpret. You can also write and edit a script and tell CINT to execute the statements in it with the .x command:

```
root[ ] .x MyScript.C
```

1.4 ACLiC: The Automatic Compiler of Libraries for CINT

Instead of having CINT interpret your script there is a way to have your scripts compiled, linked and dynamically loaded using the C++ compiler and linker. The advantage of this is that your scripts will run with the speed of compiled C++ and that you can use language constructs that are not fully supported by CINT. On the other hand, you cannot use any CINT shortcuts and for small scripts, the overhead of the compile/link cycle might be larger than just executing the script in the interpreter.

ACLiC will build a CINT dictionary and a shared library from your C++ script, using the compiler and the compiler options that were used to compile the ROOT executable. You do not have to write a makefile remembering the correct compiler options, and you do not have to exit ROOT.

To build, load, and execute a script with ACLiC you append a "++" at the end of the script file name, and use the CINT .x command.

```
root[] .x MyScript.C++
```

ACLiC executes two steps and a third one if needed. These are:

1. Calling `rootcint` to create a CINT dictionary.
2. Calling the compiler to build the shared library from the script
3. If there are errors, it calls the compiler to build a dummy executable to clearly report unresolved symbols.

1.5 First Interactive Session

In this first session, start the ROOT interactive program. This program gives access via a command-line prompt to all available ROOT classes. By typing C++ statements at the prompt, you can create objects, call functions, execute scripts, etc. Go to the directory `$ROOTSYS/tutorials` and type:

```
bash$ root
root [0] 1+sqrt(9)
(double)4.000000000000e+00
root [1] for (int i = 0; i < 5; i++) printf("Hello %d\n", i)
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
root [2] .q
bash $
```

As you can see, if you know C or C++, you can use ROOT, and there is no new command-line or scripting language to learn. To exit use `.q`, which is one of the few "raw" interpreter commands. The dot is the interpreter escape symbol. There are also some dot commands to debug scripts (step, step over, set breakpoint, etc.) or to load and execute scripts. Let's now try something more interesting. Again, start root:

```
bash$ root
root [0] TF1 f1("func1","sin(x)/x", 0, 10)
root [1] f1.Draw()
root [2] f1.Integral(0,2)
root [3] f1.Dump()
root [4] .q
```

Here you create an object of class `TF1`, a one-dimensional function. In the constructor, you specify a name for the object (which is used if the object is stored in a database), the function and the upper and lower value of x. After having created the function object you can, for example, draw the object by executing the **`TF1::Draw()`** member function.

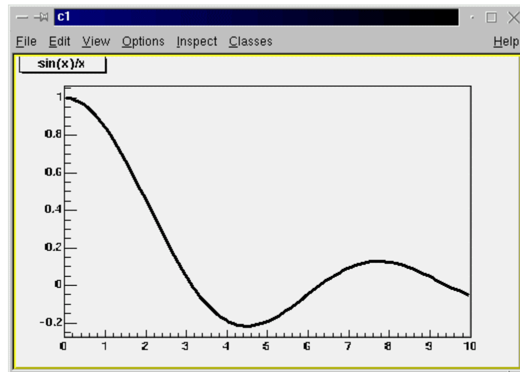


Figure 2: The result of TF1:Draw

Now, move the mouse over the picture and see how the shape of the cursor changes whenever you cross an object. At any point, you can press the right mouse button to pop-up a context menu showing the available member functions for the current object. For example, move the cursor over the function so that it becomes a pointing finger, and then press the right button. The context menu shows the class and name of the object. Select item `SetRange` and put 10,10 in the dialog box fields. (This is equivalent to executing the member function `f1.SetRange(10,10)` from the command-line prompt, followed by `f1.Draw()`.) Using the `Dump()` member function (that each ROOT class inherits from the basic ROOT class `TObject`), you can see the complete state of the current object in memory. The `Integral()` function shows the function integral between the specified limits.

1.6 The Object Browser

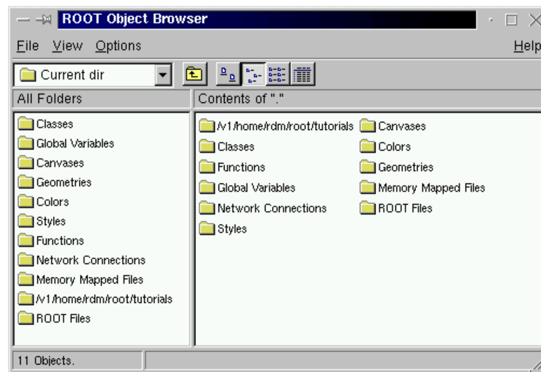


Figure 3: the Object Browser

Using the ROOT Object Browser all objects in the ROOT framework can be browsed and inspected. To create a browser object, type:

```
root [] new TBrowser
```

The browser displays in the left pane the ROOT collections and in the right pane the objects in the selected collection. Double clicking on an object will execute a default action associated with the class of the object. Double clicking on a histogram object will draw the histogram. Right clicking on an object will bring up a context menu (just as in a canvas).

2 HISTOGRAMS AND FITS

Let's start ROOT again and run the following two scripts. Note: if the above doesn't work, make sure you are in the tutorials directory.

```
bash$ root
root [] .x hsimple.C
root [] .x ntuple1.C
 // interact with the pictures in the canvas
root [] .q
```

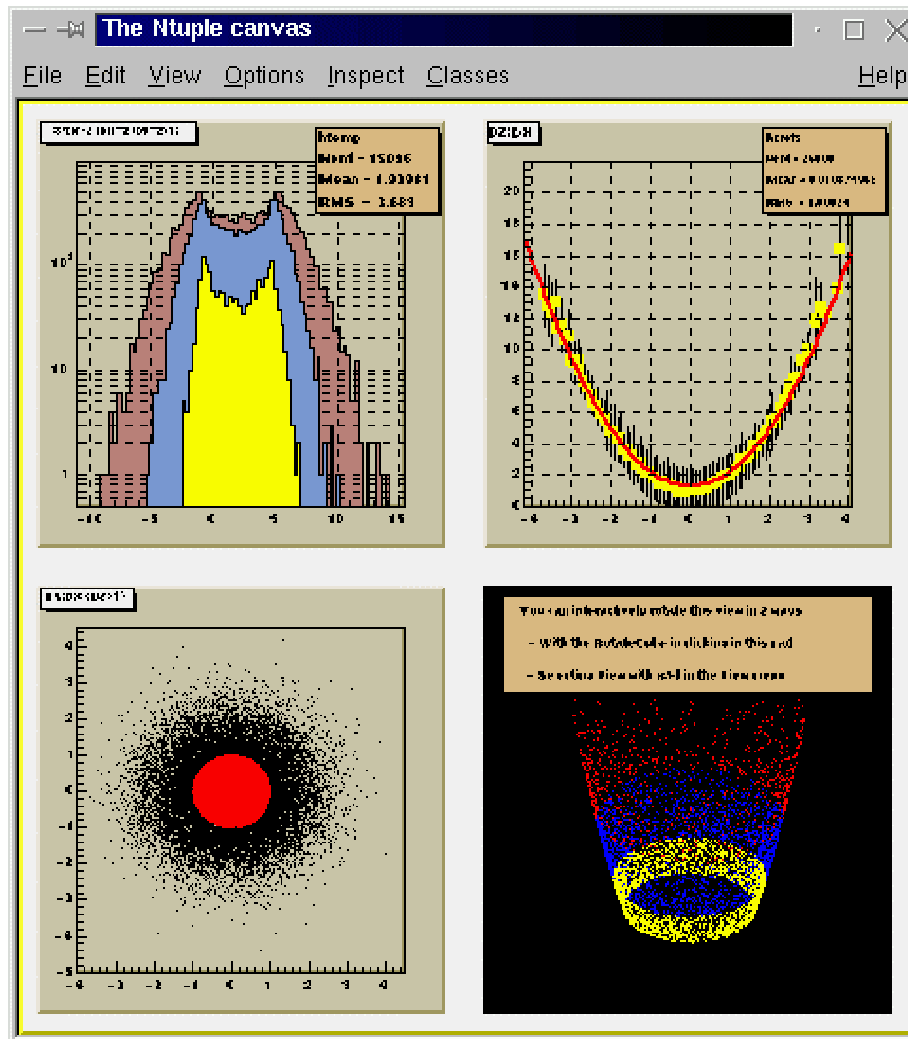


Figure 4: ROOT Histograms

Script `hsimple.C` (see `$ROOTSYS/tutorials/hsimple.C`) creates some 1D and 2D histograms and an Ntuple object. (An Ntuple is a collection of tuples; a tuple is a set of numbers.) The histograms and Ntuple are filled with random numbers by executing a loop 25,000 times. During the filling the 1D histogram is drawn in a canvas and updated each 1,000 fills. At the end of the script, the histogram and Ntuple objects are stored in a ROOT database.

The `ntuple1.C` script uses the database created in the previous script. It creates a canvas object and four graphics pads. In each of the four pads a distribution of different Ntuple quantities is drawn. Typically, data analysis is done by drawing in a histogram with one of the tuple quantities when some of the other quantities pass a certain condition. For example, our Ntuple contains the quantities `px`, `py`, `pz`,

random and i. This command will fill a histogram containing the distribution of the px values for all tuples for which pz < 1.

```
root[] ntuple->Draw("px", "pz< 1")
```

2.1 The Histogram Classes

ROOT supports the following histogram types:

1-D histograms:

- TH1C : histograms with one byte per channel. Maximum bin content = 255
- TH1S : histograms with one short per channel. Maximum bin content = 65535
- TH1F : histograms with one float per channel. Maximum precision 7 digits
- TH1D : histograms with one double per channel. Maximum precision 14 digits

2-D histograms:

- TH2C : histograms with one byte per channel. Maximum bin content = 255
- TH2S : histograms with one short per channel. Maximum bin content = 65535
- TH2F : histograms with one float per channel. Maximum precision 7 digits
- TH2D : histograms with one double per channel. Maximum precision 14 digits

3-D histograms:

- TH3C : histograms with one byte per channel. Maximum bin content = 255
- TH3S : histograms with one short per channel. Maximum bin content = 65535
- TH3F : histograms with one float per channel. Maximum precision 7 digits
- TH3D : histograms with one double per channel. Maximum precision 14 digits

Profile histograms: (TProfile and TProfile2D)

Profile histograms are used to display the mean value of Y and its RMS for each bin in X. Profile histograms are in many cases an elegant replacement of two-dimensional histograms. The inter-relation of two measured quantities X and Y can always be visualized by two-dimensional histogram or scatter-plot; If Y is an unknown (but single-valued) approximate function of X, this function is displayed by a profile histogram with much better precision than by a scatter-plot.

All histogram classes are derived from the base class TH1.

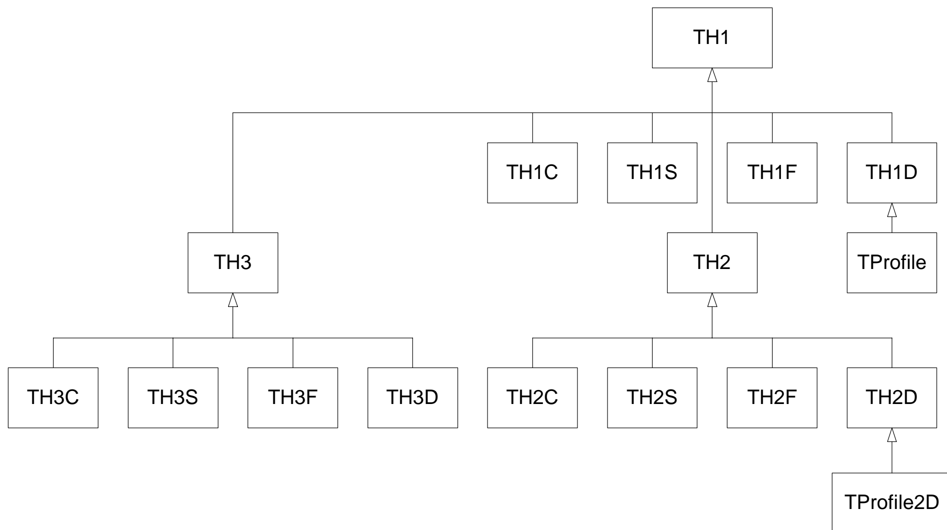


Figure 5: The Histogram Classes

The TH*C classes also inherit from the array class `TArrayC`.
The TH*S classes also inherit from the array class `TArrayS`.
The TH*F classes also inherit from the array class `TArrayF`.
The TH*D classes also inherit from the array class `TArrayD`.

2.2 Creating histograms

Histograms are created by invoking one of the constructors:

```
TH1F *h1 = new TH1F("h1","h1 title",100,0,4.4);
TH2F *h2 = new TH2F("h2","h2 title",40,0,4,30,-3,3);
```

Histograms may also be created by:

Calling the `Clone` method, see below
Making a projection from a 2-D or 3-D histogram, see below
Reading a histogram from a file

2.3 Fixed or variable bin size

All histogram types support either fixed or variable bin sizes. 2-D histograms may have fixed size bins along X and variable size bins along Y or vice-versa. The functions to fill, manipulate, draw or access histograms are identical in both cases. Each histogram always contains three objects `TAxis`: `fXaxis`, `fYaxis`, and `fZaxis`. To access the axis parameters, do:

```
TAxis *xaxis = h->GetXaxis();
Double_t binCenter = xaxis->GetBinCenter(bin);
```

See class `TAxis` for a description of all the access functions. The axis range is always stored internally in double precision.

2.4 Bin numbering convention

For all histogram types: nbins, xlow, xup

bin = 0;	underflow bin
bin = 1;	first bin with low-edge xlow INCLUDED
bin = nbins;	last bin with upper-edge xup EXCLUDED
bin = nbins+1;	overflow bin

In case of 2-D or 3-D histograms, a "global bin" number is defined. For example, assuming a 3-D histogram with binx, biny, binz, the function returns a global/linear bin number.

```
Int_t bin = h->GetBin(binx,biny,binz);
```

This global bin is useful to access the bin information independently of the dimension.

2.5 Filling Histograms

A histogram is typically filled with statements like:

```
h1->Fill(x);
h1->Fill(x,w); //with weight
h2->Fill(x,y)
h2->Fill(x,y,w)
h3->Fill(x,y,z)
h3->Fill(x,y,z,w)
```

The `Fill` method compute the bin number corresponding to the given `x`, `y` or `z` argument and increment this bin by the given weight. The `Fill` method return the bin number for 1-D histograms or

global bin number for 2-D and 3-D histograms. If TH1::Sumw2 has been called before filling, the sum of squares is also stored. One can also increment directly a bin number via TH1::AddBinContent or replace the existing content via TH1::SetBinContent. To access the bin content of a given bin, do:

```
Double_t binContent = h->GetBinContent(bin);
```

By default, the bin number is computed using the current axis ranges. If the automatic binning option has been set via: h->SetBit(TH1::kCanRebin); then, the Fill method will automatically extend the axis range to accommodate the new value specified in the Fill argument. The method used is to double the bin size until the new value fits in the range, merging bins two by two. This automatic binning options is extensively used by the TTree::Draw function when histogramming Tree variables with an unknown range. This automatic binning option is supported for 1-d, 2-D and 3-D histograms.

During filling, some statistics parameters are incremented to compute the mean value and root mean square with the maximum precision. In case of histograms of type TH1C, TH1S, TH2C, TH2S, TH3C, TH3 a check is made that the bin contents do not exceed the maximum positive capacity (127 or 65535). Histograms of all types may have positive or/and negative bin contents.

2.6 Re-binning

At any time, an histogram can be re-binned via TH1::Rebin. This method returns a new histogram with the re-binned contents. If bin errors were stored, they are recomputed during the re-binning.

2.7 Associated Error

By default, for each bin, the sum of weights is computed at fill time. One can also call TH1::Sumw2 to force the storage and computation of the sum of the square of weights per bin. If Sumw2 has been called, the error per bin is computed as the sqrt(sum of squares of weights), otherwise the error is set equal to the sqrt(bin content). To return the error for a given bin number, do:

```
Double_t error = h->GetBinError(bin);
```

2.8 Associated function

One or more object (typically a TF1*) can be added to the list of functions (fFunctions) associated to each histogram. When TF1::Fit is invoked, the fitted function is added to this list. Given an histogram h, one can retrieve an associated function with:

```
TF1 *myfunc = h->GetFunction("myfunc");
```

2.9 Operations on histograms

Many types of operations are supported on histograms or between histograms:

- Addition of a histogram to the current histogram
- Additions of two histograms with coefficients and storage into the current histogram
- Multiplications and Divisions are supported in the same way as additions.
- The Add, Divide and Multiply functions also exist to add, divide or multiply a histogram by a function.

If a histogram has associated error bars (TH1::Sumw2 has been called), the resulting error bars are also computed assuming independent histograms. In case of divisions, binomial errors are also supported.

2.10 Fitting histograms

Histograms (1-D, 2-D, 3-D, and Profiles) can be fitted with a user specified function via TH1::Fit. When a histogram is fitted, the resulting function with its parameters is added to the list of functions of this histogram. If the histogram is made persistent, the list of associated functions is also persistent. Given a pointer (see above) to an associated function myfunc, one can retrieve the function/fit parameters with calls such as:

```
Double_t chi2 = myfunc->GetChisquare();
Double_t par0 = myfunc->GetParameter(0); // value of 1st parameter
Double_t err0 = myfunc->GetParError(0);  // error on first parameter
```

2.11 Projections of Histograms

One can:

- make a 1-D projection of a 2-D histogram or Profile see functions TH2::ProjectionX,Y, TH2::ProfileX,Y, TProfile::ProjectionX
- make a 1-D, 2-D or profile out of a 3-D histogram see functions TH3::ProjectionZ, TH3::Project3D.

One can fit these projections via: TH2::FitSlicesX,Y, TH3::FitSlicesZ

2.12 Random numbers and histograms

TH1::FillRandom can be used to randomly fill an histogram using the contents of an existing TF1 function or another TH1 histogram (for all dimensions). For example the following two statements create and fill a histogram 10000 times with a default gaussian distribution of mean 0 and sigma 1:

```
TH1F h1("h1","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",10000);
```

TH1::GetRandom can be used to return a random number distributed according the contents of an histogram.

2.13 Making a copy of a histogram

Like for any other ROOT object derived from TObject, one can use the Clone method. This makes an identical copy of the original histogram including all associated errors and functions:

```
TH1F *hnew = (TH1F*)h->Clone();
hnew->SetName("hnew"); // recommended otherwise you
                       // get 2 histograms with  the same name.
```

2.14 Normalizing histograms

One can scale an histogram such that the bins integral is equal to the normalization parameter via

```
 TH1::Scale(Double_t norm);
```

2.15 Drawing histograms

Histograms are drawn via the THistPainter class. Each histogram has a pointer to its own pointer (to be usable in a multithreaded program). Many drawing options are supported. See TH1::Draw for more details.

The same histogram can be drawn with different options in different pads. When a histogram drawn in a pad is deleted, the histogram is automatically removed from the pad or pads where it was drawn. If a histogram is drawn in a pad, then filled again, the new status of the histogram will be automatically shown in the pad next time the pad is updated. One does not need to redraw the histogram.

To draw the current version of a histogram in a pad, one can use

```
h->DrawCopy();
```

This makes a clone (see `Clone` above) of the histogram. Once the clone is drawn, the original histogram may be modified or deleted without affecting the aspect of the clone. One can use `TH1::SetMaximu` and `TH1::SetMinimum` to force a particular value for the maximum or the minimum scale on the plot.

`TH1::UseCurrentStyle` can be used to change all histogram graphics attributes to correspond to the current selected style. This function must be called for each histogram. In case one reads and draws many histograms from a file, one can force the histograms to inherit automatically the current graphics style by calling before `gROOT->ForceStyle();`

The `TH1::Draw()` method has many draw options. You can combine them in a list separated by commas. For the most up to date list of the draw options please see: http://root.cern.ch/root/html/TH1.html#TH1:Draw

2.16 Setting Drawing histogram contour levels (2-D hists only)

By default, contours are automatically generated at equidistant intervals. A default value of 20 levels is used. This can be modified via `TH1::SetContour` or `TH1::SetContourLevel`. The contours level info is used by the drawing options "`cont`", "`surf`", and "`lego`".

2.17 Setting histogram graphics attributes

The histogram classes inherit from the attribute classes: `TAttLine, TAttFill, TAttMarker` and `TAttText`. See the member functions of these classes for the list of option.

2.18 Giving the axis a title

```
h->GetXaxis()->SetTitle("X axis title");
h->GetYaxis()->SetTitle("Y axis title");
```

The histogram title and the axis titles can be any `TLatex` string. The titles are part of the persistent histogram.

2.20 Saving/Reading histograms to/from a Root file

The following statements create a ROOT file and store a histogram on the file. Because `TH1` derives from `TNamed`, the key identifier on the file is the histogram name:

```
TFile f("histos.root","new");
TH1F h1("hgaus","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",10000);
h1->Write();
```

To read this histogram in another ROOT session, do:

```
TFile f("histos.root");
TH1F *h = (TH1F*)f.Get("hgaus");
```

One can save all histograms in memory to the file by:

```
file->Write();
```

2.21 Miscellaneous histogram operations

`TH1::KolmogorovTest`	statistical test of compatibility in shape between two histograms.
`TH1::Smooth`	smoothes the bin contents of a 1-d histogram

`TH1::Integral`	returns the integral of bin contents in a given bin range
`TH1::GetMean(int axis)`	returns the mean value along axis
`TH1::GetRMS(int axis)`	returns the Root Mean Square along axis
`TH1::GetEntries`	returns the number of entries
`TH1::Reset()`	resets the bin contents and errors of an histogram.

3 COLLECTION CLASSES

Collections are a key feature of the ROOT framework. Many, if not most, of the applications you write will use collections. A collection is a group of related objects. You will find it easier to manage a large number of items as a collection. For example, a diagram editor might manage a collection of points and lines. A set of widgets for a graphical user interface can be placed in a collection. A geometrical model can be described by collections of shapes, materials and rotation matrices. Collections can themselves be placed in collections. Collections act as flexible alternatives to traditional data structures of computers science such as arrays, lists and trees.

3.1 General Characteristics

The ROOT collections are polymorphic containers that hold pointers to `TObjects`, so:

1) They can only hold objects that inherit from `TObject`
2) They return pointers to `TObjects` that have to be cast back to the correct subclass

Collections are dynamic and they can grow in size as required. They are descendants of `TObject` so can themselves be held in collections. It is possible to nest one type of collection inside another to any level to produce structures of arbitrary complexity.

Collections don't own the objects they hold for the very good reason. The same object could be a member of more than one collection. Object ownership is important when it comes to deleting objects; if nobody owns the object, it could end up as wasted memory (i.e. a memory leak) when no longer needed. If a collection is deleted, its objects are not. The user can force a collection to delete its objects, but that is the user's choice.

3.2 Types of Collections

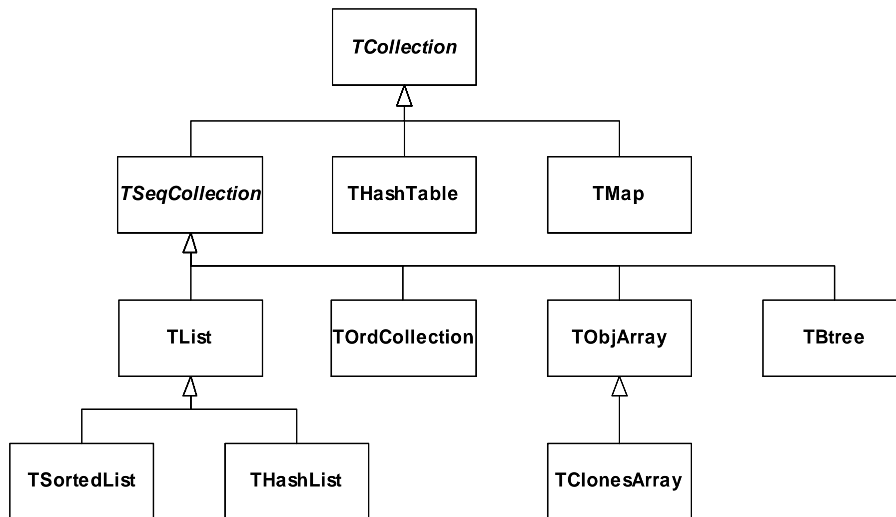


Figure 6: the Collection Classes

The ROOT system implements the following basic types of collections: unordered collections, ordered collections and sorted collections. This figure shows the inheritance hierarchy for the primary collection classes. All primary collection classes derive from the abstract base class `TCollection`.

3.3 Ordered Collections (Sequences)

Sequences are collections that are externally ordered because they maintain internal elements according to the order in which they were added. The following sequences are available: `TList`, `THashList`, `TOrdCollection`, `TObjArray`, and `TClonesArray`. The `TOrdCollection`, `TObjArray`, as well as the `TClonesArray` can be sorted using their `Sort()` member function (if the stored items are sort-able). Ordered collections all derive from the abstract base class `TSeqCollection`.

3.4 Sorted Collection

Sorted collections are ordered by an internal (automatic) sorting mechanism. The following sorted collections are available: `TSortedList and TBtree.` The stored items must be sort-able.

3.5 Unordered Collection

Unordered collections don't maintain the order in which the elements were added, i.e. when you iterate over an unordered collection, you are not likely to retrieve elements in the same order they were added to the collection. The following unordered collections are available: `THashTable and Tmap.`

3.6 Iterators: Processing a Collection

The concept of processing all the members of a collection is generic, i.e. independent of any specific representation of a collection. To process each object in a collection one needs some type of cursor that is initialized and then steps over each member of the collection in turn. Collection objects could provide this service but there is a snag: as there is only one collection object per collection there would only be one cursor. Instead, to permit the use of as many cursors as required, they are made separate classes called iterators. For each collection class there is an associated iterator class that knows how to sequentially retrieve each member in turn. The relationship between a collection and its iterator is very close and may require the iterator to have full access to the collection (i.e. it is a friend). For example: `TList TListIter`, and `TMap TMapIter`. In general, iterators will be used via the `TIter` wrapper class.

3.7 A Collectable Class

The basic protocol the `TObject` class defines for collection elements are:

`IsEqual()` Is used by the `FindObject()` collection method. By default `IsEqual()` compares the two object pointers.

`Compare()` Returns –1, 0 or 1 depending if the object is smaller, equal or larger than the other object. By default a `TObject` has not a valid `Compare()` method.

`IsSortable()` Returns true if the class is sortable (i.e. if it has a valid `Compare()` method). By default, a `TObject` is not sortable.

`Hash()` Returns a hash value. This method needs to be implemented if an object has to be stored in a collection using a hashing technique, like `THashTable`, `THashList` and `TMap`. By default `Hash()` returns the address of the object. It is essential to choose a good hash function.

3.8 The `TIter` Generic Iterator

As stated above, the `TIterator` class is abstract; it is not possible to create `TIterator` objects. However, it should be possible to write generic code to process all members of a collection so there is a need for a generic iterator object. A `TIter` object acts as generic iterator. It provides the same `Next()` and `Reset()` methods as `TIterator` but it has no idea how to support them! It works as follows:

To create a TIter object its constructor must be passes an object that inherits from TCollection. The TIter constructor calls the MakeIterator() method of this collection to get the appropriate iterator object that inherits from TIterator.

The Next() and Reset() methods of TIter simply call the Next() and Reset() methods of the iterator object. So TIter simply acts as a wrapper for an object of a concrete class inheriting from TIterator. For example:

```
MyClass *myobject;
TList *mylist = GetPointerToList();
TIter myiter(mylist);
while ((myobject = (MyClass) myiter.Next())) {
   // process myobject
}
```

4 THE GUI CLASSES

Embedded in the ROOT system is an extensive set of GUI classes. The GUI classes provide a full OO-GUI framework as opposed to a simple wrapper around a GUI such as Motif. All GUI elements do their drawing via the TGXW low-level graphics abstract base class. Depending on the platform on which you run ROOT, the concrete graphics class (inheriting from TGXW) is either TGX11 or TGWin32. All GUI widgets are created from "first principles", i.e., they only use routines like **DrawLine**, **FillRectangle**, **CopyPixmap**, etc., and therefore, the TGX11 implementation only needs the X11 and Xpm libraries. The advantage of the abstract base class approach is that porting the GUI classes to a new, non X11/Win32, platform requires only the implementation of an appropriate version of TGXW (and of TSystem for the OS interface).

All GUI classes are fully scriptable and accessible via the interpreter. This allows for fast prototyping of widget layouts. They are based on the XClass'95 library written by David Barth and Hector Peraza. The widgets have the well-known Windows 95 look and feel. For more information on XClass'95, see ftp://mitac11.uia.ac.be/html-test/xclass.html.

5 INTEGRATING YOUR OWN CLASSES INTO ROOT

In this section, we'll give a step-by-step method for integrating your own classes into ROOT. Once integrated you can save instances of your class in a ROOT database, inspect objects at run-time, create and manipulate objects via the interpreter, generate HTML documentation, etc. A very simple class describing some person attributes is shown above. The Person implementation file Person.cxx is shown here.

```
#ifndef __PERSON_H
#define __PERSON_H
#include <TObject.h>

class Person : public TObject { // need to inherit from TObject

private:
   int age;      // age of person
   float height; // height of person

public:
   Person(int a = 0, float h = 0) : age(a), height(h) { }
   int get_age(void) const { return age; }
   float get_height(void) const { return height; }

   void set_age(int a) { age = a; }
   void set_height(float h) { height = h; }

   ClassDef(Person,1) // Person class
};
#endif
```

In this section, we'll give a step-by-step method for integrating your own classes into ROOT. Once integrated you can save instances of your class in a ROOT database, inspect objects at run-time, create and manipulate objects via the interpreter, generate HTML documentation, etc. A very simple class describing some person attributes is shown above. The Person implementation file Person.cxx is shown here.

```
#include "Person.h"
// ClassImp provides the implementation of some
// functions defined in the ClassDef script
ClassImp(Person)
```

The scripts **ClassDef** and **ClassImp** provide some member functions that allow a class to access its interpreter dictionary information. Inheritance from the ROOT basic object, `TObject`, provides the interface to the database and introspection services.

Now run the **rootcint** program to create a dictionary, including the special I/O streamer and introspection methods for class `Person`:

```
bash$ rootcint -f dict.cxx -c Person.h
```

Next compile and link the source of the class and the dictionary into a single shared library:

```
bash$ g++ -fPIC -I$ROOTSYS/include -c dict.cxx
bash$ g++ -fPIC -I$ROOTSYS/include -c Person.cxx
bash$ g++ -shared -o Person.so Person.o dict.o
```

Now start the ROOT interactive program and see how we can create and manipulate objects of class Person using the CINT C++ interpreter:

```
bash$ root
root [0] gSystem->Load("Person")
root [1] Person rdm(37, 181.0)
root [2] rdm.get_age()
(int)37
root [3] rdm.get_height()
(float)1.810000000000e+02
root [4] TFile db("test.root","new")
root [5] rdm.Write("rdm") // Write is inherited from theTObject class
root [6] db.ls()
TFile** test.root
 TFile* test.root
 KEY: Person rdm;1
root [7] .q
```

Here the key statement was the command to dynamically load the shared library containing the code of your class and the class dictionary.

In the next session, we access the **rdm** object we just stored on the database `test.root`.

```
bash$ root
root [0] gSystem->Load("Person")
root [1] TFile db("test.root")
root [2] rdm->get_age()
(int)37
root [3] rdm->Dump() // Dump is inherited from the TObject class
age              37          age of person
height           181         height of person
fUniqueID        0           object unique identifier
fBits            50331648    bit field status word
root [4] .class Person
[follows listing of full dictionary of class Person]
root [5] .q
```

```
 //---------- begin fill.C
{
   gSystem->Load("Person");
   TFile *f = new TFile("test.root","recreate");
   Person *t;
   for (int i = 0; i < 1000; i++) {
      char s[10];
      t = new Person(i, i+10000);
      sprintf(s, "t%d", i);
      t->Write(s);
      delete t;
   }
   f->Close();
}
```

A C++ script that creates and stores 1000 persons in a database is shown above. To execute this script, do the following:

```
bash$ root
root [0] .x fill.C
root [1] .q
```

This method of storing objects would be used only for several thousands of objects. The special Tree object containers should be used to store many millions of objects of the same class.

```
//---------- begin find.C
void find(int begin_age = 0, int end_age = 10)
{
   gSystem->Load("Person");
   TFile *f = new TFile("test.root");
   TIter next(f->GetListOfKeys());
   TKey *key;
   while (key = (TKey*)next()) {
      Person *t = (Person *) key->Read();
      if (t->get_age() >= begin_age && t->get_age() <= end_age) {
         printf("age = %d, height = %f\n", t->get_age(), t->get_height());
      }
      delete t;
   }
}
```

This listing is a C++ script that queries the database and prints all persons in a certain age bracket. To execute this script do the following:

```
bash$ root
root [0] .x find.C(77,80)
age = 77, height = 10077.000000
age = 78, height = 10078.000000
age = 79, height = 10079.000000
age = 80, height = 10080.000000
NULL
root [1] find(888,895)
age = 888, height = 10888.000000
age = 889, height = 10889.000000
age = 890, height = 10890.000000
age = 891, height = 10891.000000age = 892, height = 10892.000000
age = 893, height = 10893.000000
age = 894, height = 10894.000000
age = 895, height = 10895.000000
root [2] .q
```

With Person objects stored in a Tree, this kind of analysis can be done in a single command.

Finally, a small C++ script that prints all methods defined in class `Person` using the information stored in the dictionary is shown below:

```
//------------ begin method.C
{
   gSystem->Load("Person");
   TClass *c = gROOT->GetClass("Person");
   TList *lm = c->GetListOfMethods();
   TIter next(lm);
   TMethod *m;
   while (m = (TMethod *)next()) {
     printf("%s %s%s\n", m->GetReturnTypeName(), m->GetName(),
             m->GetSignature());
   }
}
```

To execute this script, type:

```
bash$ root
root [0] .x method.C
class Person Person(int a = 0, float h = 0)
int get_age()
float get_height()
void set_age(int a)
void set_height(float h)
const char* DeclFileName()
int DeclFileLine()
const char* ImplFileName()
int ImplFileLine()
Version_t Class_Version()
class TClass* Class()
void Dictionary()
class TClass* IsA()
void ShowMembers(class TMemberInspector& insp, char* parent)
void Streamer(class TBuffer& b)
class Person Person(class Person&)
void ~Person()
root [1] .q
```

The above examples prove the functionality that can be obtained when you integrate, with a few simple steps, your classes into the ROOT framework.

6 OBJECT PERSISTENCY

Object persistency, the ability to save and read objects, is fully supported by ROOT. The `TFile` and `TTree` classes make up the backbone of I/O. The `TFile` is a ROOT class encapsulating the behavior of a file containing ROOT objects, and the `TTree` is a data structure optimized to hold many same class objects.

6.1 ROOT Files

A ROOT file is like a UNIX file directory. It can contain directories and objects organized in unlimited number of levels. It also is stored in machine independent format (ASCII, IEEE floating point, Big Endian byte ordering). This is an example to explain the physical layout of a ROOT file. This example creates a ROOT file and 15 histograms, fills each histogram with 1000 entries from a gaussian distribution, and writes them to the file.

```
{
   TFile f("demo.root","recreate");
   char name[10], title[20];
   for (Int_t i = 0; i < 15; i++) {
      sprintf(name,"h%d",i);
      sprintf(title,"histo nr:%d",i);
      TH1F *h = new TH1F(name,title,100,-4,4);
      h->FillRandom("gaus",1000);
      h->Write();
   }
   f.Close();
}
```

The example begins with a call to the `TFile` constructor. `TFile` is the class describing the ROOT file. In the next section, when we discuss the logical file structure, we will cover `TFile` in detail. We can view the contents by creating a `TBrowser` object.

```
root [] TFile f("demo.root")
root [] TBrowser browser;
```

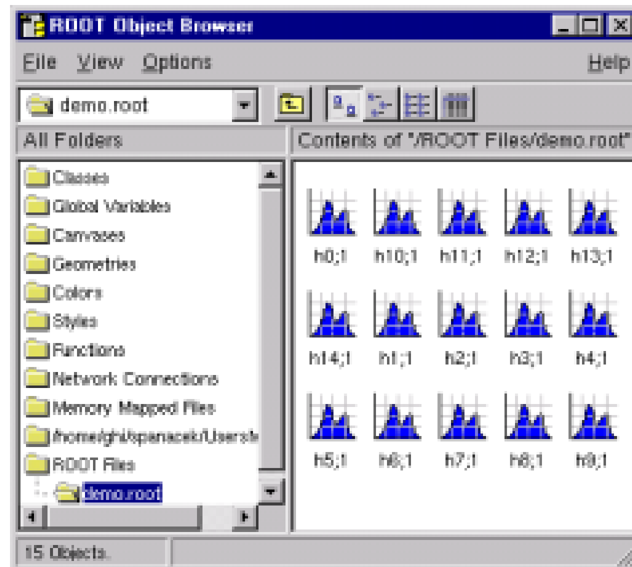


Figure 7: The Newly Created Histograms

In the browser, we can see the 15 histograms we created. Once we have the `TFile` object we can

call the `TFile::Map()` method to view the physical layout. . The first 64 bytes are taken by the file header. What follows are the 15 histograms in records of various length. Each record has a header (a `TKey`) with information about the object including its directory.

A ROOT file has a maximum size of two GB. It keeps track of the free space in terms of records and bytes. This count also includes the deleted records, which are available again.

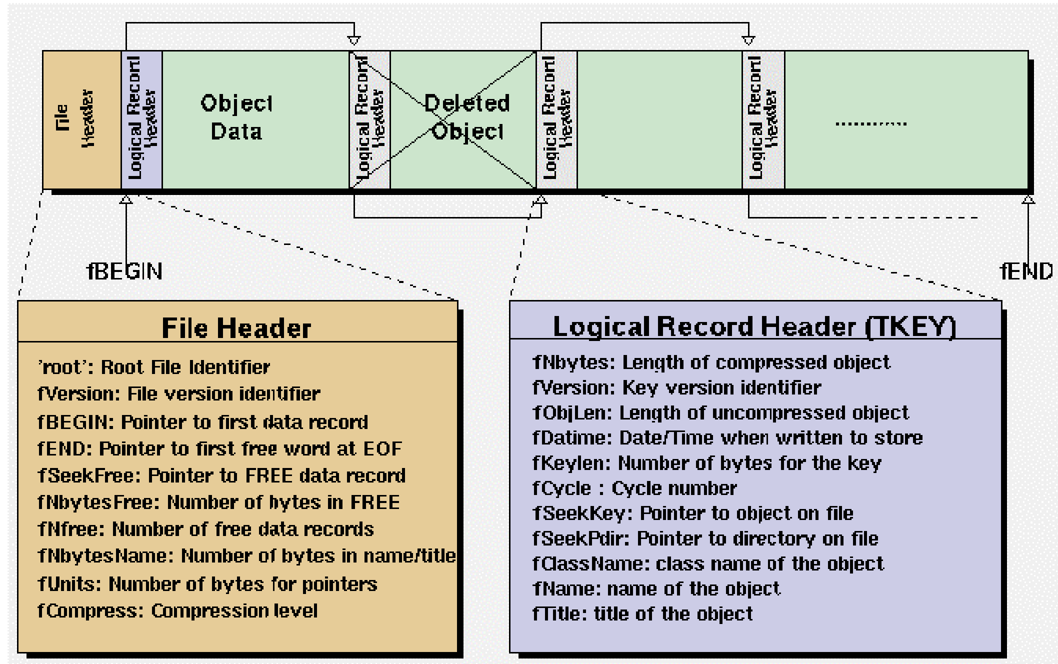


Figure 8: The ROOT file structure

6.2 File Recovery

A file may become corrupted or it may be impossible to write it to disk and close it properly. For example if the file is too large and exceeds the disk quota, or the job crashes or a batch job reaches its time limit before the file can be closed. In these cases, it is imminent to recover and retain as much information as possible. ROOT provides an intelligent and elegant file recovery mechanism using the redundant directory information in the record header.

If the file is not closed due to for example exceeded the time limit, and it is opened again, it is scanned and rebuilt according to the information in the record header. The recovery algorithm reads the file and creates the saved objects in memory according to the header information. It then rebuilds the directory and file structure. If the file is opened in write mode, the recovery makes the correction on disk when the file is closed; however if the file is opened in read mode, the correction can not be written to disk. You can also explicitly invoke the recovery procedure by calling the `TFile::Recover()` method. You must be aware of the 2GB size limit before you attempt a recovery. If the file has reached this limit, you cannot add more data. You can still recover the directory structure, but you cannot save what you just recovered to the file on disk.

6.3 Compression

The last parameter in the `TFile` constructor is the compression level. By default, objects are compressed before being written to a file. Data in the records can be compressed or uncompressed, but the record headers are never compressed. ROOT uses a compression algorithm based on the well-known gzip algorithm. This algorithm supports up to nine levels of compression, and the default ROOT uses is

one. The level of compression can be modified by calling the `TFile::SetCompressionLevel()` method. If the level is set to zero, no compression is done. Experience with this algorithm indicates a compression factor of 1.3 for raw data files and around two on most DST files is the optimum. The choice of one for the default is a compromise between the time it takes to read and write the object vs. the disk space savings. The time to uncompress an object is small compared to the compression time and is independent of the selected compression level. Note that the compression level may be changed at any time, but the new compression level will only apply to newly written objects. Consequently, a ROOT file may contain objects with different compression levels.

Compression	Bytes
0	13797
1	6290
5	6103
9	5912

The table shows four runs of the demo script that creates 15 histograms with different compression parameters.

6.4 The Logical ROOT File: TFile and TKey

We saw that the `TFile::Map()` method reads the file sequentially and prints information about each record while scanning the file. It is not feasible to only support sequential access and hence ROOT provides random or direct access, i.e. reading a specified object at a time. To do so, `TFile` keeps a list of `TKeys`, which is essentially an index to the objects in the file. The `TKey` class describes the record headers of objects in the file. For example, we can get the list of keys and print them. To find a specific object on the file we can use the `TFile::Get()` method.

```
root [] TFile f("demo.root")
root [] f.GetListOfKeys()->Print()
TKey Name = h0, Title = histo nr:0, Cycle = 1
…
TKey Name = h14, Title = histo nr:14, Cycle = 1
root [] TH1F *h9 = (TH1F*)f.Get("h9");
```

The `TFile::Get()` finds the `TKey` object with name "h9".

Since the keys are available in a `TList` of `TKeys,` we can iterate over the list of keys:

```
{
  TFile f("demo.root");
  TIter next(f.GetListOfKeys());
  TKey *key;
  while ((key=(TKey*)next())) {
    printf(
     "key: %s points to an object of class: %s at %d\n",
     key->GetName(),
     key->GetClassName(),key->GetSeekKey()
    );
  }
}
```

6.5 Viewing the Logical File Contents

`TFile` is a descendent of `TDirectory`, which means it behaves like a `TDirectory`. We can list the contents, print the name, and create subdirectories. In a ROOT session, you are always in a directory and the directory you are in is called the current directory and is stored in the global variable ***gDirectory***.

Here we open a file and list its contents:

```
root [] TFile f ("hsimple.root", "UPDATE")
root [] f.ls()
TFile** hsimple.root
TFile* hsimple.root
KEY: TH1F hpx;1 This is the px distribution
KEY: TH2F hpxpy;1 py vs px
KEY: TProfile hprof;1 Profile of pz versus px
KEY: TNtuple ntuple;1 Demo ntuple
```

It shows the two lines starting with `TFile` followed by four lines starting with the word "KEY". The four KEYs tell us that there are four objects on disk in this file. The syntax of the listing is:

```
KEY: <class> <variable>;<cycle number> <title>
```

For example, the first line in the list means there is an object in the file on disk, called `hpx`. It is of the class `TH1F` (one-dimensional histogram of floating-point numbers). The object's title is "This is the px distribution".

If the line starts with OBJ, the object is in memory. The <class> is the name of the root class (T-something). The <variable> is the name of the object. The cycle number along with the variable name uniquely identifies the object. The <title> is the string given in the constructor of the object as title.

6.6 Retrieving Objects from Disk

Multiple versions of an object with the same name, but unique cycle numbers can be in a ROOT file. ROOT automatically retrieves the one with the highest cycle number. To read an earlier version we have to explicitly specify the cycle number.

6.7 Subdirectories and Navigation

The `TDirectory` class lets you organize its contents into subdirectories, and `TFile` being a descendent of `TDirectory` inherits this ability.

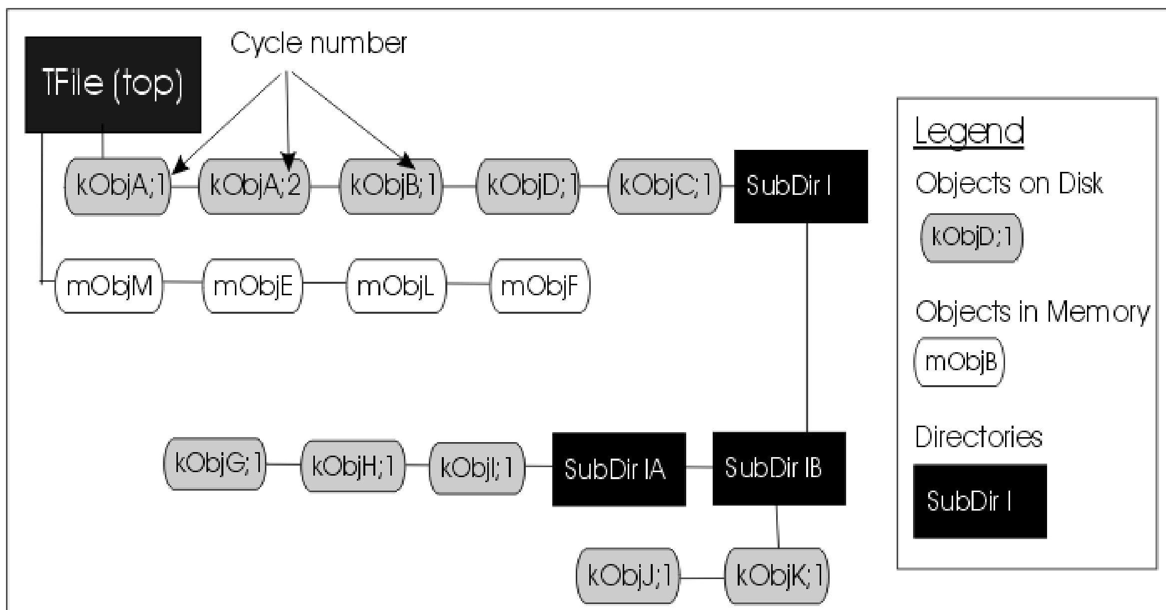


Figure 9: Objects in memory and objects on disk

This picture shows a `TFile` with five objects in the top directory (`kObjA;1, kObjA;2, kObjB;1, kObjC;1 and kObjD;1`). `ObjA` is on file twice with two different cycle numbers. It also shows four objects in memory (`mObjE, mObjeF, mObjM, mObjL`). It also shows several

subdirectories. Note that you could have the same named object, for example kOBJA , in a subdirectory and it would be unique because the entire path name is considered when identifying an object.

6.8 ROOT Trees

We explained how objects could be saved in ROOT files. In case you want to save large quantities of the same class objects, ROOT offers the TTree and TNtuple classes specifically for that purpose. The TTree class is optimized to reduce disk space and enhance access speed. A TTree can hold all kinds of data, such as objects or arrays in addition to all the simple types. A TNtuple is a TTree that is limited to only hold floating-point numbers.

We can use an example to illustrate the difference in saving individual objects vs. filling and saving the tree. Let's assume we have one million events and we write each one to a file, not using a tree. The TFile, being unaware that an event is always an event and the header information is always the same, will contain one million copies of the header. This header is about 60 bytes long, and contains information about the class, such as its name and title. For one million events, we would have 60 million bytes of redundant information. For this reason, ROOT gives us the TTree class. A TTree is smart enough not to duplicate the object header, and is able to reduce the header information to about four bytes per object, saving 56 million bytes in our case.

When using a TTree, we fill its branch buffers with data and the buffer is written to file when it is full. Branches and buffers are explained below. It is important to realize that not each object is written out individually, but rather collected and written a bunch at a time. In our example, we would fill the TTree with the million events and save the tree incrementally as we fill it. The TTree is also used to optimize the data access. A tree uses a hierarchy of branches, and each branch can be read independently from any other branch.

Assume that our Event class has 100 data members and Px and Py are two of them. We would like to compute $Px^2 + Py^2$ for every event and histogram the result. If we had saved the million events without a TTree we would have to: 1) read each event in its entirety (100 data members) into memory, 2) extract the Px and Py from the event, 3) compute the sum of the squares, and 4) fill a histogram. We would have to do that a million times! This is very time consuming, and we really do not need to read the entire event, every time. All we need are two little data members (Px and Py).

If we used a tree with one branch containing Px and another branch containing Py, we can read all values of Px and Py by only reading the Px and Py branches. This is much quicker since the other 98 branches (one per data member) are never read. This makes the use of the TTree very attractive.

6.9 Branches

A TTree is organized into a hierarchy of branches. These typically hold related variables or 'leaves'. By now, you probably guessed that the class for a branch is called TBranch. The organization of branches allows the designer to optimize the data for the anticipated use.

If two variables are independent, and the designer knows the variables will not be used together, they should be placed on separate branches. If, however, the variables are related, such as the coordinates of a point, it is most efficient to create one branch with both coordinates on it. A variable on a TBranch is called a leaf (yes - TLeaf). Another point to keep in mind when designing trees is the branches of the same TTree can be written to separate files. To add a TBranch to a TTree we call the TTree::Branch() method. The branch type differs by what is stored in them. A branch can hold an entire object, a list of simple variables, or an array of objects. Each branch has a branch buffer that is used to collect the values of the branch variable (leaf). Once the buffer is full, it is written to the file. Because branches are written individually, it saves writing empty or half filled branches.

6.10 Autosave

Autosave gives the option to save all branch buffers every n bytes. We recommend using Autosave for large acquisitions. If the acquisition fails to complete, you can recover the file and all the contents since the last Autosave. To set the number of bytes between Autosave you can use the

TTree::SetAutosave() method. You can also call TTree::Autosave in the acquisition loop every n entries.

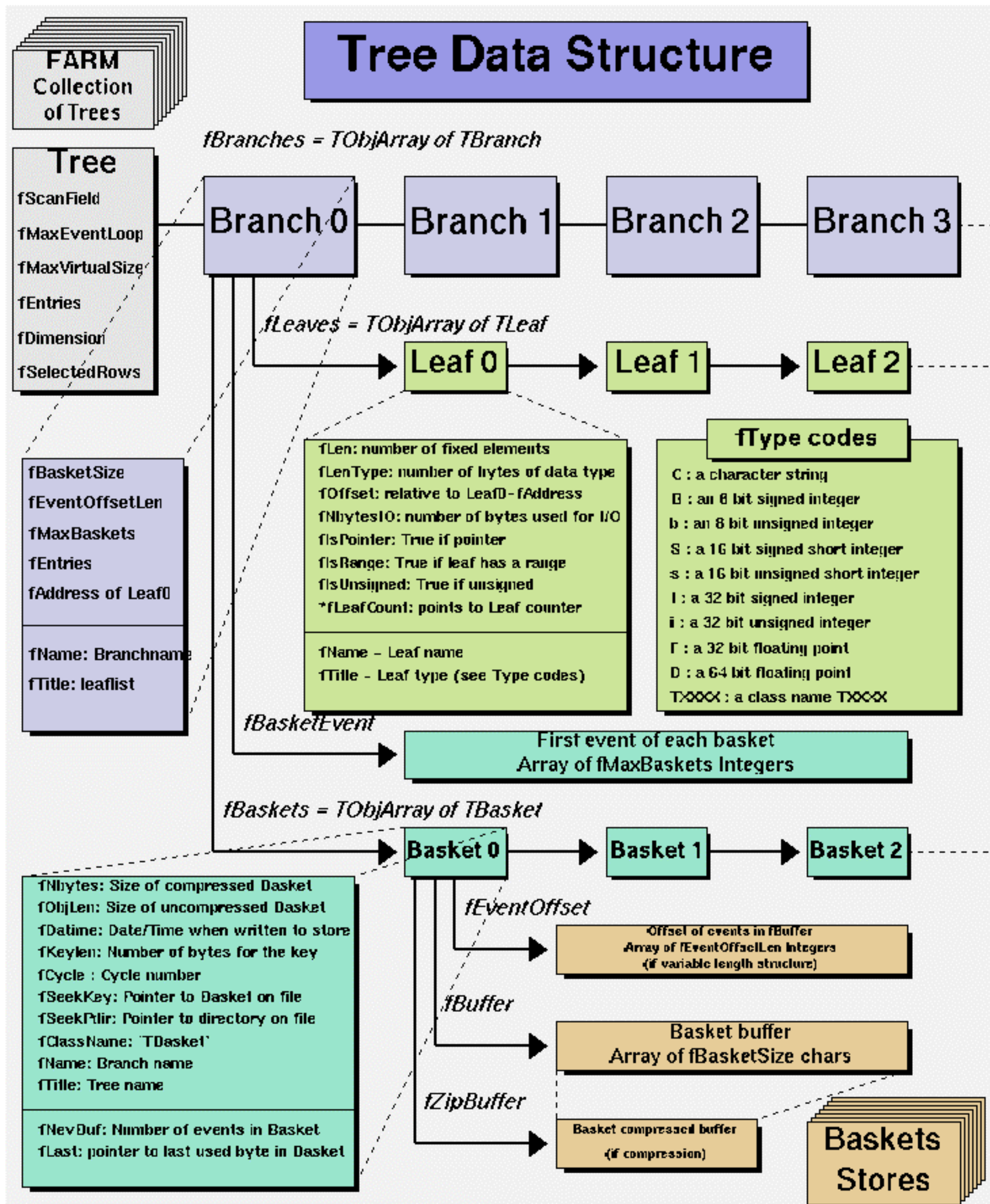


Figure 10 The Tree structure

6.11 Using Trees in Analysis

ROOT trees are not just optimized for access speed and data storage, but they also have a good set of built in analysis methods. There are several ways to analyze data stored in a ROOT Tree

Using TTree::Draw:

This is very convenient and efficient for small tasks. A TTree::Draw method produces one histogram at the time. The histogram is automatically generated. The selection expression may be specified in the command line.

Using the TTreeViewer:

This is a graphical interface to TTree::Draw with the same functionality.

Using the code generated by `TTree::MakeClass`:

In this case, the user creates an instance of the analysis class. He has the control over the event loop and he can generate an unlimited number of histograms.

Using the code generated by `TTree::MakeSelector`:

Like for the code generated by `TTree::MakeClass`, the user can do complex analysis. However, he cannot control the event loop. The event loop is controlled by `TTree::Process` called by the user. This solution is illustrated in the example analysis section at the end of this paper. The advantage of this method is that it can be run in a parallel environment using PROOF (the Parallel Root Facility).

6.12 The Tree Viewer

To bring up a tree viewer call the `TTree::StartViewer()` method. This pictureshows the Tree Viewer with several floating-point numbers leafs.

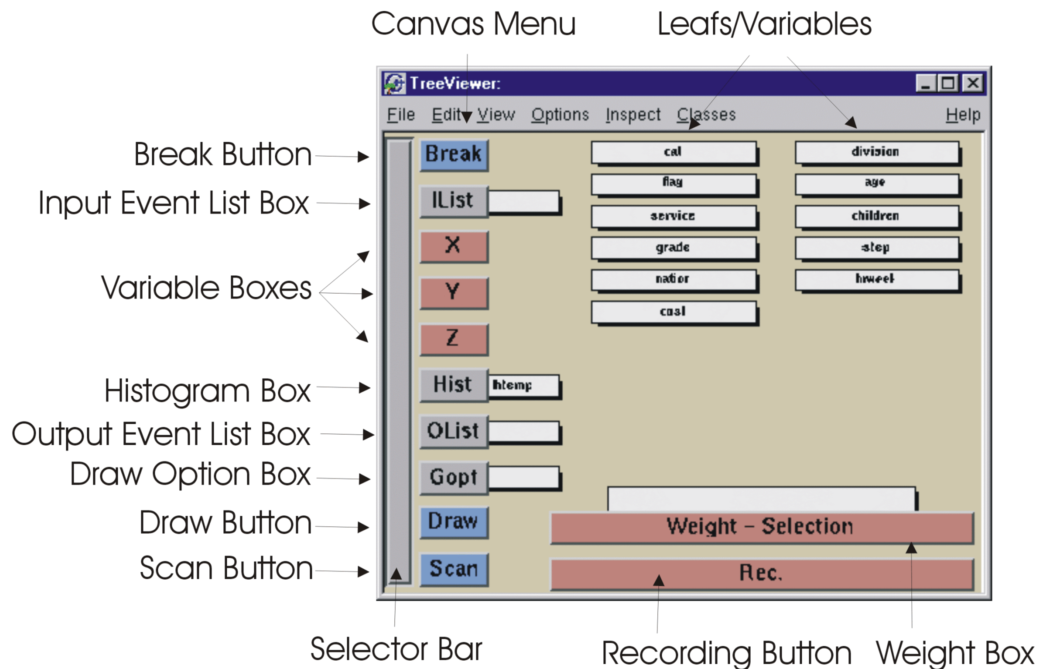


Figure 11: The Tree Viewer

The tree viewer allows you to quickly histogram any variable in the tree by double clicking on the leaf. With it, one can draw Lego plots, add cuts, select a list of events, and much more.

6.13 Simple Analysis using TTree::Draw

To show the different `Draw` options, we create a canvas with four sub-pads. We will use one sub-pad for each `Draw` command.

```
root [] TCanvas *myCanvas = new TCanvas("c","C", 0,0,600,400)
root [] myCanvas->Divide(2,2)
root [] myCanvas->cd(1)
root [] MyTree->Draw("fNtrack");
```

As you can see this signature of `Draw` has only one parameter. It is a string containing the leaf name. We activate the second pad and use this version of Draw:

```
root [] myCanvas->cd(2)
root [] MyTree->Draw("sqrt(fNtrack):fNtrack");
```

This signature still only has one parameter, but it now has two dimensions separated by a colon ("x:y"). The item to be plotted can be an expression not just a simple variable. In general, this parameter is a string that contains up to three expressions, one for each dimension, separated by a colon ("e1:e2:e3").

Change the active pad to 3, and add a selection to the list of parameters of the draw command.

```
root []myCanvas->cd(3)
root []MyTree->Draw("sqrt(fNtrack):fNtrack","fTemperature > 20.8");
```

This will draw the fNtrack for the entries with a temperature above 20.8 degrees. In the selection parameter, you can use any C++ operator, plus some functions defined in TFormula. The next parameter is the draw option for the histogram:

```
root []myCanvas->cd(4)
root []MyTree->Draw("sqrt(fNtrack):fNtrack", "fTemperature > 20.8","surf2");
```

There are many draw options. You can combine them in a list separated by commas. For the most up to date list of the draw options please see: http://root.cern.ch/root/html/TH1.html#TH1:Draw

There are two more optional parameters to the Draw method: one is the number of entries and the second one is the entry to start with.

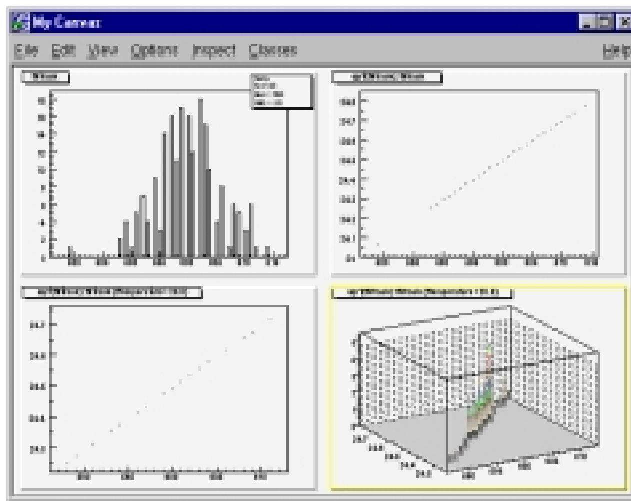


Figure 12: Draw Options

6.14 Creating an Event List

The `TTree::Draw` method can also be used to build a list of the selected entries. When the first argument is preceded by `">>"` ROOT knows that this command is not intended to draw anything, but to save the entries in a list with the name given by the first argument. The resulting list is a `TEventList`, and is added to the objects in the current directory. For example, to create a `TEventList` of all entries with more than 600 tracks:

```
root [] TFile *f = new TFile("AFile.root")
root [] T->Draw(">> myList", " fNtrack > 600")
```

This list contains the entry number of all entries with more than 600 tracks. To see the entry numbers use the `TEventList::Print("all")` method.

When using the ">>" whatever was in the TEventList is overwritten. The TEventList can be grown by using the ">>+" syntax. For example to add the entries, with exactly 600 tracks:

```
root [] T->Draw(">>+ myList", " fNtrack == 600")
```

If the Draw command generates duplicate entries, they are not added to the list.

6.15 Using an Event List

The TEventList can be used to limit the TTree to the events in the list. The SetEventList method tells the tree to use the event list and hence limits all subsequent TTree methods to the entries in the list. In this example, we create a list with all entries with more than 600 tracks and then set it so the Tree will use this list. To reset the TTree to use all events use SetEventList(0).

In the code snipped below, the entries with over 600 tracks are saved in a TEventList called myList. We get the list from the current directory and assign it to the variable list. Then we instruct the tree T to use the new list and draw it again.

```
root [] T->Draw(">>myList", " fNtrack >600")
root [] TEventList *list = (TEventList*)gDirectory->Get("myList")
root [] T->SetEventList(list)
root [] T->Draw("fNtrack ")
```

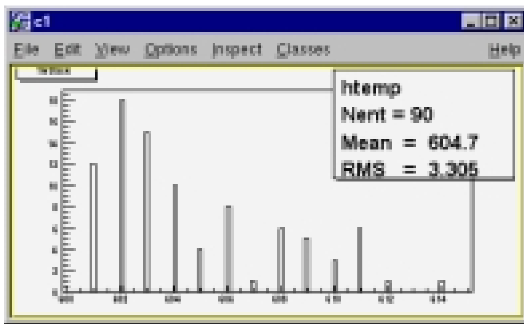


Figure 13: Histogram with the events from the Event list

6.16 Creating a Histogram

The TTree::Draw method can also be used to fill a specific histogram. The syntax is:

```
root [] TFile *f = new TFile("AFile.root")
root [] T->Draw("fNtrack >> myHisto")
root [] myHisto->Print()
TH1.Print Name= myHisto, Total sum= 200
```

As we can see, this created a TH1, called myHisto. If you want to append more entries to the histogram, you can use this syntax:

```
root [] T->Draw("fNtrack >>+ myHisto")
```

If you want to fill a histogram, but not draw it you can use the TTree::Project() method.

```
root [] T->Project("quietHisto","fNtrack")
```

6.17 Tree Information

Once we have drawn a tree, we can get information about the tree. These are the methods used to get information from a drawn tree:

- GetSelectedRows: Returns the number of entries accepted by the selection expression.

In case where no selection was specified, it returns the number of entries processed.

- GetV1: Returns a pointer to the double array of the first variable.
- GetV2: Returns a pointer to the double array of second variable
- GetV3: Returns a pointer to the double array of third variable.
- GetW: Returns a pointer to the double array of weights where the weight equals the result of the selection expression.

For a complete description and examples of these methods see:

http://root.cern.ch/root/html/TTree.html#TTree:Draw

6.18 More Complex Analysis using TTree::MakeClass

The Draw method is convenient and easy to use, however it falls short if you need to do some programming with the variable.

For example, for plotting the masses of all oppositely changed pairs of tracks, you would need to write a program that loops over all events, finds all pairs of tracks, and calculates the required quantities. We have shown how to retrieve the data arrays from the branches of the tree in the previous section, and you could write that program from scratch. Since this is a very common task, ROOT provides two utilities that generate a skeleton class designed to loop over the entries of the tree. The TTree::MakeClass and the TTree::MakeSelector methods. TTree::MakeSelector is explained and used in the next section with an example analysis.

6.19 Creating a Class with MakeClass

We load the shared library libEvent.so to make the class definitions available, and open the ROOT file. In the file, we see the TTree, and we use the TTree::MakeClass method on it. MakeClass takes one parameter, a string containing the name of the class to be made. In the command below, the name of our class will be “MyClass”.

```
root [] .L libEvent.so
root [] TFile *f = new TFile ("Event.root");
root [] f->ls();
TFile**          Event.root      TTree benchmark ROOT file
 TFile*          Event.root      TTree benchmark ROOT file
  KEY: TH1F      htime;1 Real-Time to write versus time
  KEY: TTree     T;1     An example of a ROOT tree
  KEY: TH1F      hstat;1 Event Histogram
root [] T->MakeClass("MyClass")
Files: MyClass.h and MyClass.C generated from Tree: T
(Int_t)0
```

The call to MakeClass created two files. MyClass.h contains the class definition and several methods, and MyClass.C contains MyClass::Loop. The .C file is kept as short as possible, and contains only code that is intended for customization. The .h file contains all the other methods. Here is a description of each method.

- MyClass(TTree *tree=0): This constructor has an optional tree for a parameter. If you pass a tree, MyClass will use it rather than the tree from witch it was created.
- Void Init(TTree *tree): Init is called by the constructor to initialize the tree for reading. It associates each branch with the corresponding leaf data member.
- ~MyClass(): This is the destructor, nothing special.
- Int_t GetEntry(Int_t entry): This loads the class with the entry specified. Once you have executed GetEntry, the leaf data members in MyClass are set to the values of the entry. For

example, `GetEntry(12)` loads the 13th event into the event data member of MyClass (note that the first entry is 0). `GetEntry` returns the number of bytes read.

- `Int_t LoadTree(Int_t entry)and void  Notify():`
  These two methods are related to chains. `LoadTree` will load the tree containing the specified entry from a chain of trees. `Notify` is called by `LoadTree` to adjust the branch addresses.
- `void Loop():` This is the skeleton method that loops through each entry of the tree. This is interesting to us, because we will need to customize it for our analysis.

6.20 Customizing the Loop

Here we see the implementation of `MyClass::Loop().MyClass::Loop` consists of a for-loop calling `GetEntry` for each entry. In the skeleton, the numbers of bytes are added up, but it does nothing else. If we were to execute it now, there would be no output.

```
void MyClass::Loop()
{
   Int_t nentries = Int_t(fTree->GetEntries());
   Int_t nbytes = 0, nb = 0;
   for (Int_t i=0; i<nentries;i++) {
      if (LoadTree(i) < 0) break;
      nb = fTree->GetEntry(i);   nbytes += nb;
   }
}
```

For analysis, we modify this method. For example, here we select the first 100 tracks of each event and fill a histogram with it.

```
void MyClass::Loop()
{
 Track *track  = 0; Int_t n_Tracks = 0;
  TH1F *myHisto    = new TH1F("myHisto","fPx", 100, -5,5);
  TH1F *smallHisto = new TH1F("small","fPx", 100, -5,5);
  for (Int_t i=0; i<nentries;i++) {
     if (LoadTree(i) < 0) break;
     GetEntry(i); n_Tracks = event->GetNtrack();
      for (Int_t j = 0; j < n_Tracks; j++){
        track = (Track*) event->GetTracks()->At(j);
        myHisto->Fill(track->GetPx());
        if (j < 100){ smallHisto->Fill(track->GetPx()); }
     }
  }
  myHisto->Draw();
  smallHisto->Draw("Same");
}
```

Loading MyClass

To run the analysis the new class (`MyClass`) needs to be loaded and instantiated.

```
root [] .L libEvent.so
root [] .L MyClass.C
root [] MyClass m
```

Now we can call the methods of `MyClass`. For example, we can get a specific entry. In the code snipped below, we get entry 0, and print the number of tracks (594). Then we get entry 1 and print the number of tracks (597).

```
root [] m.GetEntry(0)
(int)1
root [] m.event->GetNtrack()
(Int_t)594
root [] m.GetEntry(1)
(int)48045
root [] m.event->GetNtrack()
(Int_t)597
root [] m.Loop()
```

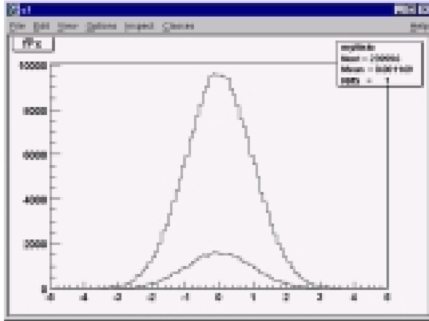


Figure 14: Results of the event loop

7 PROOF: ROOT'S PARALLEL PROCESSING FACILITY

Building on the experience gained from the implementation and operation of the PIAF system we have developed the parallel ROOT facility, PROOF. The main problems with PIAF were because its proper parallel operation depended on a cluster of homogenous equally performing and equally loaded machines. Due to PIAF's simplistic portioning of a job in N equal parts, where N is the number of processors, the overall performance was governed by the slowest node. The running of a PIAF cluster was an expensive operation since it required a cluster dedicated solely to PIAF. The cluster could not be used for other types of jobs without destroying the PIAF performance.

In the implementation of PROOF, we made the slave servers the active components that ask the master server for new work whenever they are ready. In the scheme the parallel processing performance is a function of the duration of each small job, packet, and the networking bandwidth and latency. Since the bandwidth and latency of a networked cluster are fixed the main tunable parameter in this scheme is the packet size. If the packet size is too small the parallelism will be destroyed by the communication overhead caused by the many packets sent over the network between the master and the slave servers. If the packet size is too large, the effect of the difference in performance of each node is not evened out sufficiently.

Another very important factor is the location of the data. In most cases, we want to analyze a large number of data files, which are distributed over the different nodes of the cluster. To group these files together we use a chain. A chain provides a single logical view of the many physical files. To optimize performance by preventing huge amounts of data being transferred over the network via NFS or any other means when analyzing a chain, we make sure that each slave server is assigned a packet, which is local to the node. Only when a slave has processed all its local data will it get packets assigned that cause remote access. A packet is a simple data structure of two numbers: begin event and number of events. The master server generates a packet when asked for by a slave server, taking into account t the time it took to process the previous packet and which files in the chain are local to the lave server. The master keeps a list of all generated packets per slave, so in case a slave dies during processing, all its packets can be reprocessed by the left over slaves.

8 USEFUL LINKS

The ROOT website is at: http://root.cern.ch
Various papers and talks on ROOT: http://root.cern.ch/root/Publications.html
Documentation on each ROOT class: http://root.cern.ch/root/html/ClassIndex.html
Documentation on IO: http://root.cern.ch/root/InputOutput.html

9 SUPPORTED PLATFORMS AND COMPILERS

ROOT is available on these platform/compiler combinations:

- Intel x86 Linux (g++, egcs and KAI/KCC)
- Intel Itanium Linux (g++)
- HP HP-UX 10.x (HP CC and aCC, egcs1.2 C++ compilers)
- IBM AIX 4.1 (xlc compiler and egcs1.2)
- Sun Solaris for SPARC (SUN C++ compiler and egcs)
- Sun Solaris for x86 (SUN C++ compiler)
- Compaq Alpha OSF1 (egcs1.2 and DEC/CXX)
- Compaq Alpha Linux (egcs1.2)
- SGI Irix (g++ , KAI/KCC and SGI C++ compiler)
- Windows NT and Windows95 (Visual C++ compiler)
- Mac MkLinux and Linux PPC (g++)
- Hitachi HI-UX (egcs)
- LynxOS
- MacOS (CodeWarrior, no graphics)

INTRODUCTION TO THE ANAPHE/LHC++ SOFTWARE SUITE

Andreas Pfeiffer
CERN, Geneva, Switzerland

Abstract

The Anaphe/LHC++ project is an ongoing effort to provide an Object-Oriented software environment for future HEP experiments. It is based on standards-conforming solutions, together with HEP-specific extensions and components. Data persistence is provided by the Objectivity/DB Object Database (ODBMS), while the visualisation is based on Qt (for 2-D presentation) and OpenInventor (for 3-D). To complement the standard package, a set of C++ class libraries for histogram management, Ntuple-like analysis (based on Objectivity/DB) and for presentation graphics (based on Open Inventor) have been developed. A new tool for physics data analysis, named Lizard, has been developed based on a set of abstract interfaces (as defined by the AIDA project). Its implementation is based on the python scripting language and the existing C++ class libraries of Anaphe/LHC++.

1. INTRODUCTION

The Anaphe/LHC++ project was set up to replace the full suite of functionality formerly provided by the FORTRAN based CERNLIB [1] in the new object-oriented computing environment required by (mainly) the LHC experiments.

The aim of the project is to provide a flexible, interoperable, customisable set of interfaces, libraries and tools, re-using existing (public domain or commercial) packages wherever applicable and possible. HEP specific adaptions are written where needed. In close collaboration with the experts from the experiments and other common HEP projects (like Geant-4 [2] and CLHEP [3]), these packages are designed considering the huge data volume expected for LHC, the distributed computing necessary to analyse the data as well as long-term evolution and maintenance.

The Anaphe/LHC++ [4] software suite consists of a set of individual class libraries for data storage (Objectivity/DB [5] and HepODBMS [6]), fundamental numerical mathematics (NAG-C [7] and CLHEP) and physics data analysis (HTL [8] for histogramming, Gemini/HepFitting [9] for minimization/fitting) which exist since about 1997 in their present form.

Recently, the development of packages for visualisation (Qplotter, based on the Qt [10] package) and a command-line driven interactive data analysis tool (Lizard) has started. The first release of these is scheduled for mid October 2000.

2. THE LIBRARIES

2.1 Data Storage

2.11 Packages

In order to study possible solutions for the storing and handling of the multi-Petabytes of data expected from the LHC experiments, a R&D project (RD45 [11]) was set up in 1995. The requirements included not only to store the raw data, but also the reconstructed objects, calibration and monitoring data and analysis objects like histograms and ntuples. After some study, it was found that the best candidate to fulfil the requirements is an Object Database Management Group (ODMG) [12] compliant database used

together with a mass storage system (based on the IEEE reference model [13]). Several tests on available Object Databases were performed and resulted in the choice of Objectivity/DB for the time being.

On top of this, a small (about 15 kLines) s/w layer (HepODBMS) has been developed to simplify the possible porting to a different Database implementation and to provide a higher level interface than the one specified by ODMG. The HepODBMS class libraries provide classes to deal with session management, clustering hints, tag (ntuples) and event collections.

2.12 Special features

The use of an OO Database with transaction safety (locking) guarantees consistency in the datasets written. This is especially of importance in a distributed/concurrent environment.

Another important feature of the OO Database is location independency, meaning that moving the individual database files to other file systems and/or other hosts has no impact on the user's code.

2.13 Scaling to LHC

Meanwhile, significant experience in using Objectivity has been gained and scaling behaviour has been verified by HEP experiments like BaBar and CMS.

During the last year, the BaBar experiment has accumulated more than 100 TByte of data in Objectivity/DB and by this showing scaling behaviour to amounts of data which are only one order of magnitude lower than the ones expected for LHC. Another aspect of scaling is the number of processes used in processing (filtering and reconstruction), where BaBar is presently using 140 parallel processes for online reconstruction. This again is only about one order of magnitude lower than the requirements for LHC. Of course, one order of magnitude is still a "qualitative" step, nevertheless it shows that the underlying concept scales to quite a large fraction of the size needed for LHC.

In the CMS experiment several Data Challenges have been (and are) done where a few hundred processes were reading Geant simulation data, overlaying events and reconstructing the combined events in parallel. The simulated data is read from and the reconstructed objects are stored into Objectivity/DB. The overall sustained data rate achieved is about 70 MB/s. Analysis of this (reconstructed) data is done in parallel by a few tens of physicists using several hundred jobs on a distributed farm of PCs.

2.2 Fundamental physics and mathematics functionality

For fundamental physics and mathematics functionality, like dealing with 3- and 4-Vectors, simple and complex matrix operations and optimized numerical algorithms for special mathematical functions, the CLHEP and NAG libraries are used. CLHEP, a project started in 1992, aims at providing a fundamental set of C++ classes specialized for use in HEP, while the Numerical Algorithms Group (NAG) provides an optimized set of C functions dealing with a broad spectrum of mathematical (and physical) algorithms, e.g., for minimization, complex number functions, Fourier transforms, quadrature/integration, ordinary differential equations, curve and surface fitting, linear algebra, approximations of Special Functions (Bessel et al.) and more.

2.3 Histogramming

The Histogram Template Library (HTL) is a C++ class library that provides powerful histogramming functionality. As the name suggests, it exploits the template facility of C++ and is designed to be compact, extensible, modular and performant. As such it only deals with histograms - i.e. binned data - and not unbinned or "Ntuple" data. Furthermore, although simple file-based I/O and "lineprinter" output are supported via helper classes, it is decoupled from more advanced I/O and visualisation techniques. In the context of LHC++, such capabilities are provided by other components that fully interoperate with HTL.

HTL itself offers the histogramming features of HBOOK as well as a number of useful extensions, with an object-oriented (O-O) approach. This package replaces the histOOgrams package - an earlier C++ class library for histograms. The major functional innovation over the previous package are the support for different kinds of bins, the support of both persistent and transient (i.e. in-memory) histograms at runtime and the definition of an abstract histogram interface. As a result, it is possible to work with transient histograms and subsequently save some or all of them in a database as persistent histograms in a very simple and natural way (thus simulating so called explicit I/O). This clearly has significant performance advantages, particularly in the area of filling operations.

The definition of an abstract histogram interface allows functionality that is provided by external packages, such as plotting or fitting, to be decoupled from the actual implementation of the histogram. This feature paves the way for co-existence of different histogram packages that conform to the abstract interface.

2.4 Minimizing and Fitting

2.41 Minimizing

Minimization and Fitting in the context of Anaphe/LHC++ is presently in a major re-design phase. For a detailed description of the new design, see [14]. As the new packages make extensive re-use of the previous ones, the latter are described in some detail here.

Gemini is a GEneral MINImization and error analysis package implemented as a C++ class library. Minuit's functionality is provided in a 'Minuit-style' (even if, internally, another minimizer may actually do the work) and new functionality offered by NAG C minimizers is added. Gemini thus provides a unified C++ API both to standard Minuit and to NAG C family of minimizers. For the common subset of functionality, it is up to the user which minimization engine does the work: Minuit or NAG C. The user can easily switch between various minimizers without essential changes in the application code. The currently supported set of minimizers (Minuit and NAG C) can be extended without essential changes in the API.

2.42 Fitting

HEPFitting is a collection of C++ fitting classes, based on the general minimization package Gemini. It provides an object oriented application programming interface, which allows for loading data, defining a model and a fitting criterion, performing a fit in a specified region and obtaining fit results, including error analysis.

Histogram data to be fitted to can be loaded using the HTL package. Alternatively, general data points of the form (x, y) and the corresponding errors can be loaded via user's arrays, where y stands for an experimental value measured at a point $x = (x1, ..., xp)$, and a user model $y = f(x1, ..., xp|\mathbf{a})$ can be fitted to the loaded data, resulting in an estimate of the parameter vector $\mathbf{a}$.

2.5 Visualisation

2.51 2-D Visualisation

In spring 2000, the decision was made to change the visualisation of 2-D data to use the Qt package instead of the previously used OpenInventor package. Qt is a multi-platform C++ GUI toolkit, produced by Troll Tech, it is supported on all major variants of Microsoft Windows and Unix/X Windows. Qt is the basis of the popular KDE desktop environment on Linux.

On top of this package, a small set of additional functionality (specific for HEP) has been added in the form of the Qplotter package. The Qplotter library allows a programmer to produce graphic representation of physics data (such as histograms, scatter plots or curves) both on the screen and as PostScript files. The same package could be used to produce simple 2D drawings (e.g. test beam setup),

but it will not provide directly event display features, since they can be implemented using 3D packages like OpenInventor/OpenGL. Nevertheless the library should eventually allow mixing both kind of images in the same viewer.

Full user interactivity (such as point-and-click or drag-and-drop) will not be pursued from the beginning, although it may be eventually supported by add-on tools.

2.52 3-D Visualisation

The low-level 3D graphics is based on the OpenGL library, the de-facto industry standard. The OpenGL rendering process is highly optimised in order to obtain satisfactory performance. In general, OpenGL is implemented in low-cost hardware accelerating graphics cards. The high-level 3D graphics is based on OpenInventor [15], which is a high-level C++ class library based on OpenGL.

The foundation concept in Inventor is the ”scene database” which defines the objects to be used in an application. When using Inventor, a programmer creates, edits, and composes these objects into hierarchical 3d scene graphs (i.e., database). A variety of fundamental application tasks such as rendering, picking, event handling, and file reading/writing are built-in operations of all objects in the database and thus are simple to invoke.

Since Inventor is object-oriented (written in C++), it encourages programmers to extend the system by writing new objects. Inventor users have created a variety of new objects that are not included in the product, such as: Bezier surfaces, animation objects, special viewers, and many more.

Although presently none of the other packages provided by Anaphe/LHC++ is making use of 3-D graphics (OpenInventor), we provide and maintain OpenInventor for users who want to use it.

3. ABSTRACT INTERFACES FOR DATA ANALYSIS – THE AIDA PROJECT

The goals of the AIDA project [16] are to define abstract interfaces for common physics analysis tools, such as histograms. The adoption of these interfaces make it easier for users to select and use different tools without having to learn new interfaces or the need to change their code. In addition it should be possible to exchange data between AIDA compliant applications. The developers of the following packages and analysis systems are actively contributing to, interested in, or plan to adopt AIDA: COLT, JAS, Lizard, Open Scientist.

A set of categories has been identified for which *independent abstract interfaces will be defined. These components contain the basic data structures used in physics data analysis like Histograms, Ntuples, Functions, Vectors as well as higher level components like Fitter, Plotter, Analyzer, Event-Display and Controller/Hub.*

At the time of writing this report, this concept of abstract interfaces in combination with a number of different actual implementations has been proven and verified. In collaboration with GEANT-4 examples of simulations have been set up using the AIDA Histogram interfaces creating and filling histograms which were implemented using three different analysis systems (JAS, Lizard, Open Scientist) without the user-code being aware of it.

4. INTERACTIVE DATA ANALYSIS TOOL – THE LIZARD PROJECT

Lizard [17] is a new Interactive Analysis Environment created within the Anaphe/LHC++ context at CERN. The aim of the Lizard project is to produce an analysis tool which can be easily integrated in a C++ based environment and which provides functionalities at least comparable with the core of PAW.

4.1 Architecture and Design

Lizard is aiming at being a highly modular, flexible, interoperable and customizable tool. It is based on the set of components identified for AIDA (see above). Until all the interfaces have been agreed within

the AIDA group, a separate set of interfaces has been defined. This set will be subsequently replaced by the ones defined in AIDA as they become available.

The modularity and flexibility is achieved by a loose coupling between the individual components through their abstract interfaces. By doing this, each component is only allowed to access another through its (abstract) interface, therefore no knowledge about the implementation is needed at compile or link time (the latter due to the use of shared libraries).

This approach also ensures a high degree of customizability, as an experiment can implement their own version of any component (defined through the abstract interface). In the same way, components from other analysis systems can be used by simply loading the corresponding set of shared libraries from the other system (and possibly some "bridging code", as in the case of JAS).

4.2 Scripting

On request of the users, Lizard is using a command line driven approach. To implement this, the choice was made to use one of the various existing scripting languages as the main use cases showed to be more similar to the typical ones for scripting (repetition, macros) rather than (algorithmic) programming.

In order to keep maximal flexibility, it was decided to use SWIG [18] to "map" the commands (which are defined as a set of C++ classes with their methods) into one of several possible target scripting languages. The choice of the scripting language was Python [19] as it seemed to be the most likely one to be accepted and the fact that it is object oriented by design.

4.3 Status of Lizard

A first prototype with rather limited functionality of Lizard has been released in Feb. 2000. This prototype was basically intended to check the possibilities of the automatic "mapping" of commands into python, as well as gaining user feed-back regarding the choice of Python.

In October 2000 a first version of Lizard has been released as scheduled. Work is continuing to improve the functionality and add more features. Further releases are planned in intervals of about 4 weeks.

The study of the impact of a distributed computing environment on data analysis - as it is expected for LHC - will start in 2001. The results of this study will then be integrated into Lizard. The aim here is to create a tool which can do parallel data analysis using a large "farm" of machines (on typically a few TeraByte of data), or "small scale" analysis on a single machine (typically several GigaByte of data) with basically the same "front-end" user interface.

The "farming" environment is expected to be based on the GRID infrastructure which is presently under study and development at CERN.

5. FUTURE PLANS

5.1 Libraries

The main work on the libraries will be the implementation of wrappers to the abstract interfaces as used in AIDA and Lizard. This is done using existing class libraries wherever possible.

5.2 Lizard

Future work on Lizard can be split into the following two groups:

Near future:

- work on documentation
- direct use of Python objects (functions, list, ...)
- more plotting features

- “license free” version of Lizard
- reading of HBOOK files (histograms and ntuples)
- history recall across sessions
- XML format for histograms and vectors
- finalise Fitter interface

Longer term:

- analyse/design/implementation of distributed computing using Globus (GRID)
- allow communication between components using an Observer pattern (“live histograms”)
- more plug-in-like style to allow for several different instances of the various components to be present at a given time

5.3 AIDA

The main activity within the AIDA project will be to define and consolidate the interfaces to the other components. In the longer term, more functionality will be added, for example to allow for “live” histograms which are useful in online and monitoring environments.

6. Conclusion

The Anaphe/LHC++ software suite provides a flexible, interoperable, customisable set of interfaces, libraries and tools for physics data analysis. It is based on standards-conforming solutions, together with HEP-specific extensions and components. Data persistence is provided by the Objectivity/DB Object Database (ODBMS), while the visualisation is based on Qt (for 2-D presentation) and OpenInventor (for 3-D).

In close collaboration with the experts from the experiments and other common HEP projects, these packages are designed considering the huge data volume expected for LHC, the distributed computing necessary to analyse the data as well as long-term evolution and maintenance.

A new tool for physics data analysis, has been developed based on a set of abstract interfaces (as defined by the AIDA project). Its implementation is based on the Python scripting language and the existing C++ class libraries of Anaphe/LHC++.

In the future, we will - besides further evolution of the existing packages - start to investigate the impact of distributed computing on physics data analysis with the aim to make this type of analysis as transparent as possible for the user of the libraries and tools.

References

[1] CERN Program Library (CERNLIB); see http://wwwinfo.cern.ch/asd/index.html

[2] Geant-4 - A toolkit for the simulation of the passage of particles through matter. see http://wwwinfo.cern.ch/asd/geant4/geant4.html

[3] CLHEP - Class Libraries for HEP; see http://wwwinfo.cern.ch/asd/lhc++/clhep/index.html

[4] LHC++; see http://wwwinfo.cern.ch/asd/lhc++/index.html

[5] Objectivity/DB; see http://www.objectivity.com

[6] HepODBMS; see http://www???????

[7] Numeric Algorithms Group; see http://www.nag.co.uk

[8] HTL; see http://wwwinfo.cern.ch/asd/lhc++/htlguide/htl.html

[9] Gemini/HepFitting; see http://wwwinfo.cern.ch/asd/lhc++/Gemini http://wwwinfo.cern.ch/asd/lhc++/HepFitting

[10] Qt (Qt is a trademark of Troll Tech); see http://www.troll.no

[11] RD45 - a persistent object manager for HEP; see http://wwwinfo.cern.ch/asd/rd45/index.html

[12] The Object Data Management Group (ODMG); see http://www.odmg.org

[13] The IEEE Storage Systems Standards Working Group; see http://www.ssswg.org

[14] J.T.Moscicki, "Minimization and Fitting in Anaphe", these proceedings.

[15] Open Inventor 3d Toolkit; see http://www.tgs.com/Products/openinv-index.html

[16] AIDA - Abstract Interfaces for Data Analysis; see http://wwwinfo.cern.ch/asd/lhc++/AIDA/index.html http://aida.freehep.org

[17] Lizard; see http://wwwinfo.cern.ch/asd/lhc++/Lizard/index.html

[18] SWIG - Simplified Wrapper and Interface Generator; see http://www.swig.org

[19] The Python scripting language; see http://www.python.org

DATA STORAGE AND ACCESS IN LHC++

Marcin Nowak
CERN IT Division, RD45 project - CH 1211 Geneva 23 Switzerland

Abstract
This paper presents LHC data requirements as well as some features of HEP data models and explains how an ODBMS can be used to address them. Essential features of object databases will be discussed, followed by those specific to Objectivity/DB, which is the database currently used in LHC++. The differences between transient and persistent data models will be given with some rules for how to convert the former into the latter. Next, the paper will focus on HepODBMS layer, which is a set of HEP specific classes extending the functionality of a database and forming an interface used by other LHC++ software components. The concept of event collections and object naming will be discussed.

1. INTRODUCTION

Experiments at the Large Hadron Collider (LHC) at CERN will generate huge quantities of data: roughly 5 petabytes (10^{15} bytes) per year and about 100 PB over the whole data-taking period (15+ years). Data will be collected at rates exceeding 1GB/s and later analyzed, perhaps many times. The analysis frameworks of the new experiments will be developed using object-oriented (OO) technologies and consequently their data will be represented in object-oriented data models, often of significant complexity.

These factors form a challenging data storage and management problem and it seems clear that the traditional solutions based on sequential Fortran files would not be adequate. In 1995 the RD45 project was initiated at CERN to investigate new solutions and technologies. The emphasis was put on commercial products, with the hope of minimizing development costs and maintenance effort over the very long period of use. The evaluation of different technologies such as language extensions for persistency, light-weight object managers, object request brokers and object databases led to the recommendation of an Object Database Management System as the main data management product, together with a Mass Storage System to provide physical storage.

Studies of the various ODBMS products on the market, particularly with respect to their ability to satisfy LHC data management requirements, resulted in the selection of a particular database: currently Objectivity/DB.

Experiment	Data Rate	Data Volume
ALICE	1.5 GB/sec	1 PB/year (during one month)
CMS	100 MB/sec	1 PB/year
ATLAS	100 MB/sec	1 PB/year
LHC-B	20 MB/sec	200 TB/year

Table 1 Expected Data Rates and Volumes at LHC

2. OBJECT DATABASES

2.1 Data Model

In OO programming style the data is represented as a set of objects interconnected with each other in various ways, depending on the object model. Figure 1 shows a simple example of the data model for a HEP Event.

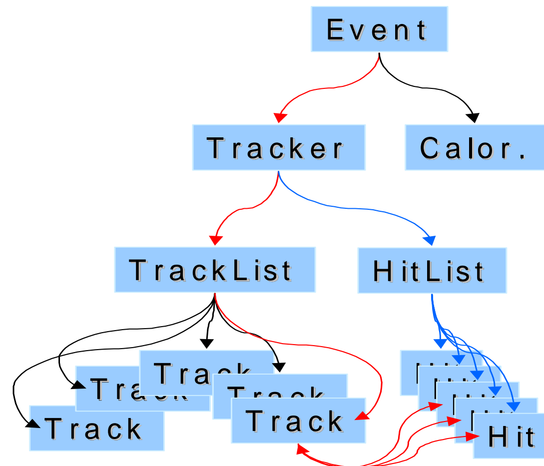


Figure 1: Simple Logical Model of Event Data Objects

An application working with a given data model would traverse the connections between objects to find the data it needs or to modify their contents. It may also modify the network of objects by adding or removing some of them. In the Event model example, the application could navigate from the main Event object to its tracking detector, retrieve a particular track from the track list, then navigate to all associated hit objects to perform a track refit.

2.2 Transient Objects

In the traditional run-cycle, an application would first create in memory the objects it needs and then fill them with some data. Next, it would perform the actual task it was designed for: working with its data representation. Finally, the program would store the results and delete objects from memory. In this scenario, the lifetime of an object is rather short and always bound to the application execution time – the objects exist only within the context of a single program. In the database terminology such objects are called *transient* (i.e. temporary).

OO languages support transient objects and navigation between them (the latter via pointers and references in C++). Creating and traversing in-memory networks of objects is very efficient and type-safe even for polymorphic classes. This, however, assumes that the entire network of objects is maintained in the memory. There is little support from today's languages regarding I/O operations on such networks of objects.

2.2.1 Object Input / Output

Providing I/O for complex data models is a difficult task for the programmer. In the first place, 2 different data formats have to be maintained for every class that is to be stored, namely:

- Class definition used by the application, including pointers to other objects
- Data encoding format used when storing in a file

The formats must be assumed to be different, as the run-time format is tightly coupled to the operating system and even compiler version. Thus, even if we start with an exact memory copy in a file, the possibility of handling different run-time formats must be provided: the application code has

to perform conversions between the two representations. The problem increases further since any individual class definition is likely to change over the long run period of LHC, leaving some objects in the old format stored on tape.

The programmer has also to decide:

- How to perform the conversion.
  The conversion of object attributes may require byte-swapping or alignment adjustment, which is a time-consuming, but rather straightforward operation. What is more difficult is storing the connections between objects, which constitute the shape of the object network. This requires translating pointers and references into a storable format and a special code that will be able to rebuild the network later.
- When to perform the I/O.
  All data transfers have to be initiated explicitly. Typically, some amount of data has to be read from disk when the application starts and all useful results have to be stored at the end. During the execution time, additional I/O operations may be required when the program follows a link referencing an object that is not yet in memory. In a multi-user environment, part of the data in memory may become stale as a result of an update performed – by another user or process – upon the corresponding object on disk. Such situations must be detected to avoid data corruption.
- How much data to transfer.
  In a complex HEP application it is difficult to predict which data items will be used. In many cases all of the event data is read, just in case it is needed. This approach may result in degraded performance.

Code that deals with object I/O often constitutes a large part of the application. Maintaining this code is both tedious and error-prone. Consistency between the disk and memory representation is not performed automatically and errors in this layer may lead to obscure software problems. In addition, large amounts of I/O related code in a class makes programs less understandable and may obscure the actual services provided by the class.

2.3 Object Persistency

Persistent objects are the opposite of transient objects. They do not disappear when the application finishes (they *persist*). This is possible because they do not belong to the application context, but rather to the context of a persistent store. In the case of an ODBMS, they belong to a database context. A persistent object will disappear only when explicitly deleted from the store.

Programs working with persistent objects do not "own" them – they receive only a copy from the store. It is possible for more then one program to access the same object at the same time in a "read" mode.

Object databases maintain the consistency between objects on disk and in memory. The programmer never deals with the disk representation – but sees only the run-time definition of the class. This feature is called "tight language binding". The ODBMS also takes care of all I/O that has to be performed to retrieve an object. All the problems discussed in section 2.2.1 are handled by the system and not by the application programmer.

Persistent objects are real objects. They support run-time type identification, (multiple) inheritance, polymorphism (virtual functions) and abstract data types. In C++ they can also be instances of templated classes.

2.4 Transactions

Object databases provide transactions in a similar way that relational databases do. The transactions are atomic, consistent, isolated and durable (so-called A.C.I.D. [2] properties) and are usually not nested. All data access is done inside a transaction – otherwise the store is not accessible. The

standard transaction types are "read" and "update". Some systems provide additional types of transactions that e.g. allow simultaneous read and write to the same objects An example of such a transaction type is the multiple reader, one writer (MROW) transaction supported by Objectivity/DB.

As all data access occurs inside a transaction, all I/O operations are transaction bound. At the start of a transaction, only the connection to the database is established. As the application proceeds to navigate in the data network and access objects, the relevant pieces of data are retrieved. The ODBMS tries to ensure that there are no unneeded data transfers, in order to optimise performance. If the application modifies objects or creates new ones, the changes may be kept in memory or written to disk, but they are not immediately visible to other clients. Only when the transaction is committed, all modifications are flushed to disk and registered in the database.

Transactions in database systems are the main tool to ensure data consistency. If a transaction is interrupted (aborted) in the middle, the database status is not changed.

2.5 Navigational Access

As described above, the main method of finding an object in the network is by navigation. Transient objects use pointers and references as links. A pointer is a memory address and uniquely identifies an object in the application context (or virtual memory address space). Persistent objects, which exist in the database context, need a different kind of identification.

When a new persistent object is created, the ODBMS assigns to it a unique Object Identifier (OID). The actual implementation of the OID varies between different systems, but they have common functionality – they allow the object to be found in the disk store. OIDs that point directly to the object are called physical and OIDs that use indirection are called logical. Logical OIDs give more flexibility at the cost of performance and scalability.

Object Identifiers replace pointers and references in persistent objects. They are used to create uni-directional (pointing in one direction, like a C++ pointer) associations between them. In most products they also enhance the idea of pointers by allowing:

- bi-directional associations
  bi-directional association is a relation between 2 objects. From an implementation point of view it may look just like 2 objects pointing to each other, but the ODBMS makes sure that pointers on both sides are set correctly (or reset) at the same time. It is not possible to modify only one of them, thus ensuring consistency.

- 1-to-n associations
  1-to-n association is a relation between one object and an arbitrary number of objects on the other side. It may be uni- or bi-directional.

The OID is typically hidden from the programmer by wrapping it in a *smart pointer* implementation. Smart pointers are small objects that behave semantically in the same way as normal pointers, but they also provide additional functionality. If the smart pointer provided by ODBMS is dereferenced (in C++ by using "*" or "->" operator on it) the system is able to check if the object pointed to is already in memory, and if not, read it from disk using the OID contained in the smart pointer. After that, the smart pointer behaves just like a normal pointer. All this happens without any additional code in the application.

The ODMG standard [1] defines ODBMS smart pointer as a templated class d_Ref<T>. Figure 2 shows an example program using d_Ref<> in the same way as normal C++ pointer.

```
Collection<Event> events;           // an event collection
Collection<Event>::iterator evt;    // a collection iterator

// loop over all events in the input collection
for(evt = events.begin();  evt != events.end(); evt++)
      {
      // access the first track in the tracklist
      d_Ref<Track> aTrack;
      aTrack  = evt->tracker->tracks[0];

      // print the charge of all its hits
      for (int i = 0; i < aTrack->hits.size(); i++)
            cout  << aTrack->hits[i]->charge
                << endl;
      }
```

Figure 2: Navigation using a C++ program

As a consequence of the tight binding of ODBMS to the programming language the application programmer perceives the database store as a natural extension to application memory space. Using the database one can create networks of objects much larger than would be possible in memory, with indefinite lifetime and the possibility to efficiently navigate among them.

2.6 Database Schema

If the ODBMS is to be able to perform automatic conversion between object representation on disk and in memory, it has to have detailed information about the object. It has to know the type, name and position of every attribute in the object. This information needs to be registered in the database before any object of a given class can be stored. All class definitions known to the ODBMS are called the *database schema*.

The schema registration process depends on the ODBMS and on the programming language. In Objectivity/DB a C++ class is entered into the schema by a program that pre-processes the header files. The headers may contain normal C++ classes, with the exception that object associations should replace pointers.

2.7 Concurrent Access to Data

ODBMS products provide support for multiple clients working on the same data store and concurrently updating it. Usually ODBMSs introduce a central “lockserver” that co-ordinates the updates by keeping a lock table for the whole system. To ensure data consistency in the system, all data changes are part of a transaction. If a transaction accesses part of the database, this region is locked with an appropriate lock mode (read, write or MROW). Subsequent clients trying to operate on the same region must first contact the lockserver to determine what type of access is allowed at a given time. All locks that a transaction has acquired last until the end of the transaction (either by commit or abort).

Locking and transactions are the mechanisms that allow concurrent access to a data store. Without them it would not be possible to guarantee data consistency.

3. CHOOSING ODBMS FOR LHC++

The next section describes specific features of Objectivity/DB - the ODBMS system that the RD45 project currently recommends as the data storage and management product for LHC experiments. The following list mentions requirements that were considered the most important for the selection:

- Standard compliance – ODMG [1]
  The use of a standards compliant API may make it easier to replace one ODBMS component with another system, if such need arises
- Scalability to hundreds of PB
- Mass Storage System interface
  LHC experiments will require a database able to store 100 PB of data, a large part of which will have to be kept in MSS (on tapes)
- Distributed system
- A centralised system will not be able to efficiently deal with such large amounts of data and serve many client applications accessing it concurrently
- Heterogeneous environment
  Research institutes have very diverse computing environments – a system that will be used by all of them should be interoperable between most of them
- Data replication
  Replicating the most frequently used data to remote institutes may have a big impact on performance
- Schema versioning
  The system should allow changes in the class definitions that will inevitably happen in the long run period of LHC
- Language heterogeneity
  LHC++ is written in C++, but there are graphical presentation tools implemented in Java that would profit from direct access to the database
- Object versioning
  This feature is used by various applications, such as a detector calibration database package

4. OBJECTIVITY/DB SPECIFIC FEATURES

This chapter focuses on specific features of Objectivity/DB.

4.1 Federations of Distributed Databases

The Objectivity/DB ODBMS supports a so-called *federation* of distributed databases. Each database within a federation corresponds to a filesystem file and may be located on any host on the network. There is one central federation (FDB) file containing the catalogue of all databases and the class schema. Hosts on which database files are located run the Objectivity data server (ooams). In addition, there is a central lockserver program located on a selected machine.

Client applications may use one Objectivity federated database at a time. To access the data within a federation, the database client software first reads the FDB catalogue to find where the data is located and then connects directly to the data server on a machine hosting the right database.

Before any data is read or modified, the client contacts the lockserver to obtain a lock. These operations are all performed transparently to the user, who only deals with (networks of) objects. Figure 3 shows an example of a Federated Database with 2 client applications accessing it from different hosts.

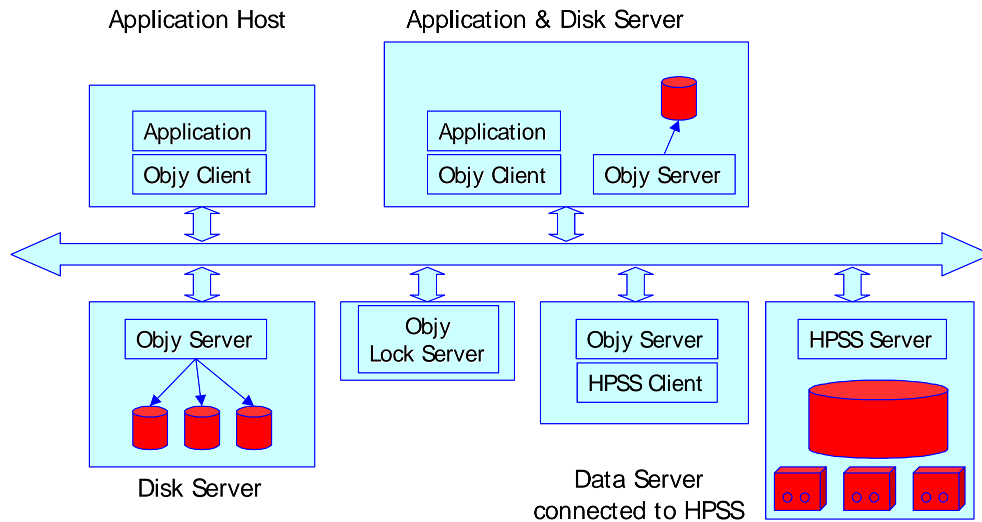


Figure 3 Distributed Applications Sharing a Federated

4.2 Physical Store Implementation

All ODBMS products use a multilevel hierarchy to implement the possibly distributed physical store. Objectivity/DB uses a hierarchy of five different levels. The topmost level - the Federated Database - keeps the catalogue of physical location of all databases that constitute the federation. Each database is structured internally into “containers” - contiguous areas of objects within a database file. Containers consist themselves of database “pages” – regions of fixed size determined at the federation creation time. Every page has “slots” for actual object data (but objects larger then a single page are allowed). Figure 4 illustrates the physical storage hierarchy in Objectivity/DB.

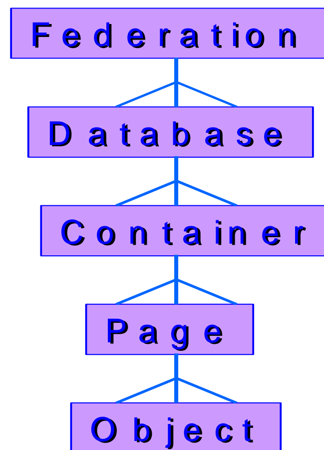


Figure 4 Storage Hierarchy in Objectivity/DB

The structure of the physical store hierarchy is directly reflected by the internal structure of the OID implementation. A 4-tuple of 16-bit numbers that represent database, container, page and slot number is used to uniquely references any object within the store.

Figure 5 Object Identifier Implementation used by Objectivity/DB

4.2.1 Separation of Logical and Physical Storage Model

The concept of OIDs allows any object to be accessed directly in the potentially large distributed store without requiring the application programmer to know the details of the store implementation, such as file and host names. Since information about the physical layout of the store is kept in a central place by the ODBMS, it is much easier to change the storage topography without compromising existing applications. One may change the location of a particular file to a new host by moving the data and changing the catalogue entry. Since the catalogue is shared by all database applications, they will use the data from the new location without any modifications.

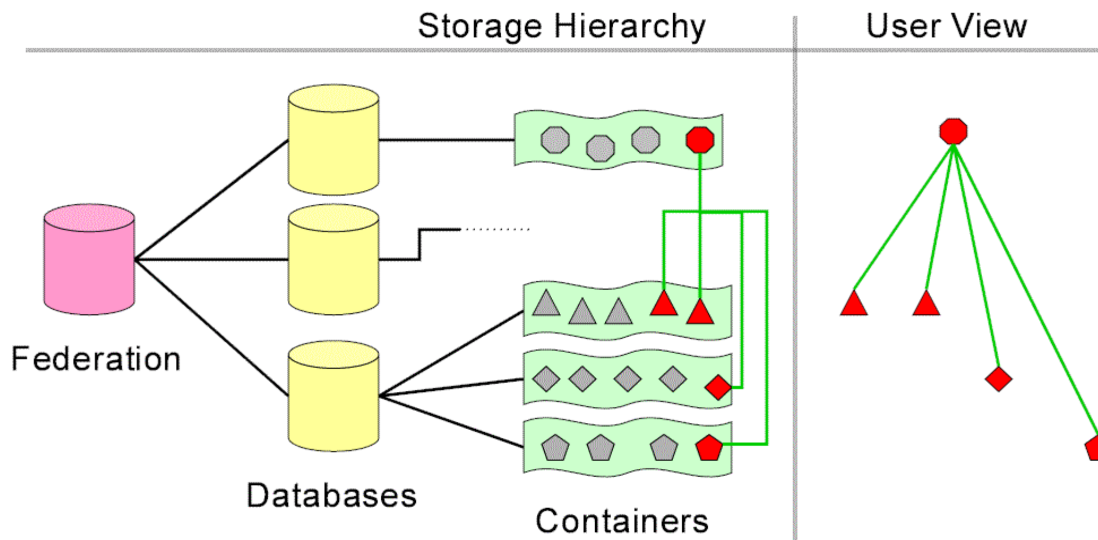


Figure 6: Physical Storage Hierarchy and Logical User View

4.2.2 Data Clustering and Re-Clustering

An important feature offered by several ODBMS products is the support for object clustering. When a persistent object is created, the programmer may supply information where the object should be placed within the physical storage hierarchy. In C++ a clustering hint may be passed as an argument to the new operator. For example, the statement

```
d_Ref<Track> aTrack = new(event)  Track;
```

instructs the database to create a new persistent track object physically close to the event object. This ability to cluster data on the physical storage medium is very important for optimising the performance of applications which access data selectively.

The goal of this clustering optimisation is to transfer only useful data from disk to the application memory (or one storage level below: from tape storage to a disk pool). Grouping data close together that will later be read together can drastically reduce the number of I/O operations needed to acquire this data from disk or tape. It is important to note that this optimisation requires some knowledge about the relative contributions of different access patterns to the data.

An simple clustering strategy is the "type based clustering" where all objects of some particular class are placed together: e.g. Track and Hit objects within an event may be placed close to each other since both classes will often be used together during the event reconstruction.

For physics analysis this simple approach is probably not very efficient since the selection of data that will be read by a particular analysis application depends more on the physics process. In this case one may group the analysis data for a particular physics process together.

4.3 Data Replication

Objectivity/DB supports the replication of all objects in a particular database to multiple physical locations. The aim of this data replication is twofold:

- To enhance performance:
  Client programs may access a local copy of the data instead of transferring data over a network.
- To enhance availability:
  Clients on sites which are temporarily disconnected from the full data store may continue to work on the subset of data for which local replicas are available.

Figure 7 shows a simple configuration where one database is replicated from site 1 to two other remote sites over a wide area network.

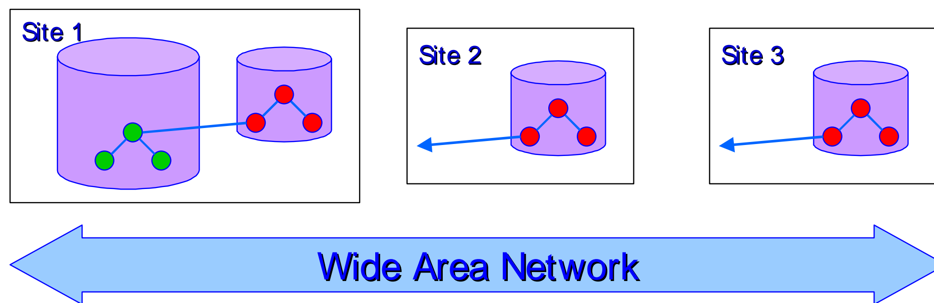


Figure 7 Database Replication

Any state changes of replicated objects on either site are transparently propagated to all other replicas by the database system. In the case that some of the replicas are not reachable, a quorum-based mechanism is used to determine which replica may be modified and a backlog of all changes is kept until other replicas become online again.

The data replication feature is expected to be very useful, for example to distribute central event selection data to multiple regional data centres.

4.4 Schema Generation

The schema generation for C++ classes in Objectivity/DB is performed using a pre-processor program (see Figure 8). The program scans class definitions of persistent classes in Objectivity's Data Definition Language (DDL) and generates C++ header and implementation files for persistent classes. The generated header files define the class interface for clients of a persistent class. The generated implementation files contain C++ code which implements smart-pointer types and various collection iterators for each persistent class. All generated files are then compiled together with any

other application code and linked against the Objectivity library to form a complete database application. The database schema is stored centrally in the federation file.

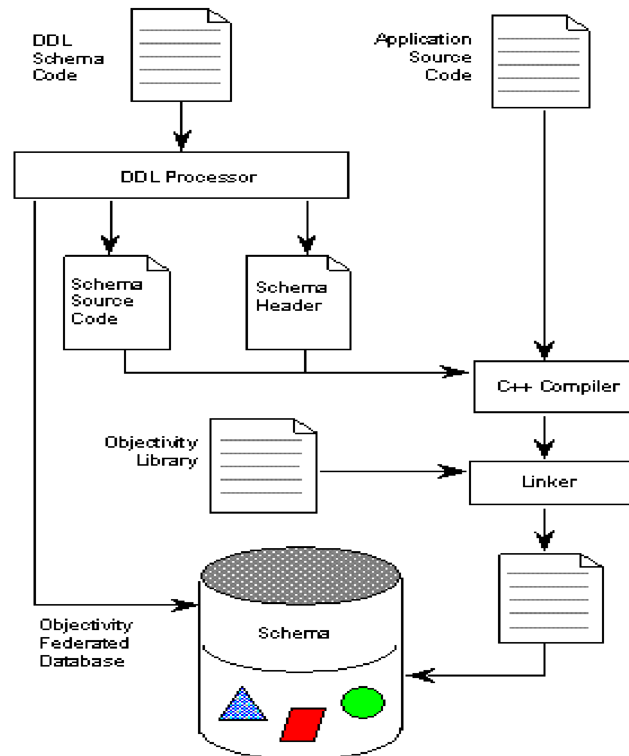


Figure 8 Schema Capture and Build Process

4.5 Creating Persistent Classes

The Data Definition Language used by Objectivity/DB is the C++ programming language extended with object associations. This makes it easy to convert transient applications and eliminates the need to learn a new programming language.

Classes become persistent by inheriting from d_Object class:

```
Class Event : public d_Object { ... };
```

Persistent classes should not contain pointers – memory pointers are meaningless in the persistent store address space. They should be replaced by references to persistent objects:

~~Event *event_pointer; // Event is transient~~

```
d_Ref<Event>  event_reference;   //Event is persistent
```

The “event_reference” follows the same semantic rules as the C++ pointer “event_pointer”.

The notion of pointers is further enhanced with 1-to-n and bi-directional associations. Below, the “events” attribute is a set of object references:

```
d_Ref<Event>  events[];
```

Bi-directional association is a two-directional link between objects. It has to be declared in both classes, but modification to it is an atomic operation that changes the values on both ends at the same time:

```
d_Ref<Event>  event <-> tracker;    // in Tracker
```

```
d_Ref<Tracker>  tracker <-> event;    // in Event
```

4.5.1 Persistent STL

There are some standard C++ classes that contain and use pointers internally, such as all STL containers. These classes can not be used directly with a database. Objectivity provides a special version of STL that can be used in persistent objects. The names of classes are the standard ones preceded by “d_”, e.g. d_vector, d_map. Here is an example declaration of a vector of Events:

```
d_vector<Event>   my_events;
```

4.6 Object Naming

The normal way of working with a network of objects is navigation. However, the navigation has to start somewhere! Objectivity/DB allows any given object to be named and later located using this name. Objects can be named in different scopes:

- on the global level of the federation
- in scope of database or a container
- in scope of any other persistent object

Using different scopes enables the creation of personal namespaces.

4.7 Object Collections

It is very common to group objects into collections. Collections can be physical, logical or a mix of the two:

- Physical grouping is achieved by placing objects into one of the physical containers or databases of the federated database. The size of the collection is then limited by the size of the physical container it is located in.
- Logical collection is a group of references to persistent objects. The references may be stored in one of the container classes, such as a vector. The size of the collection is limited by the capacity of the collection class.
- Mixed collection is a logical collection of physical containers. The size of such a collection is practically infinite.

5. HEPODBMS LAYER

HepODBMS is a software layer that is located between the ODBMS and all other LHC++ modules. Its two main functions are to provide insulation from the database API and HEP specific storage classes.

5.1 API Independence

During the lifetime of the LHC, new versions of commercial components will be released and maybe even new products will be adopted. To make transitions between them easier, the dependence on the API of a specific vendor should be minimized. This can be achieved by using standard compliant products. However, many software products use a proprietary API that makes most efficient use of their internal architecture or are simply not fully standard compliant.

In Objectivity/DB, the structure of the federated database does not exactly reflect the ODMG database - for example, there is no notion of federation or containers in the ODMB standard. Hence, the API that deals with them is non-standard. HepODBMS tries to minimize dependence on these non-standard features by providing naming indirection and providing a higher-level database session control class.

5.2 API Enhancements

5.2.1 Database Session Control

HepODBMS contains a session control class **HepDbApplication** that provides:

- Easy transaction handling
- Methods to create databases and containers and to find them later by name
- Job and transaction level diagnostics
- The ability to set options through environmental variables

Figure 9 shows an example of a simple application using the HepDbApplication class to initialize the connection to a federated database, start a transaction and create a histogram.

```
Main(){
     HepDbApplication dbApp; // create an appl. Object
     dbApp.init("MyFD");     // init FD connection dbApp.startUpdate();
     // update mode transaction
     dbApp.db("analysis");   // switch to db "analysis"

  // create a new container
     ContRef histCont = dbApp.container("histos");
  // create a histogram in this container
     HepRef(Histo1D) h = new(histCont) Histo1D(10,0,5);

     dbApp.commit();      // Commit all changes
     }
```

Figure 9 Setting up a DB session using the HepDbApplication class

5.2.2 Object Clustering

The "new" operator generated by Objectivity for each persistent class accepts an additional parameter – the so-called clustering hint described above. Any other persistent object, container or database may serve as a clustering hint. The ODBMS will attempt to place the new object as close to the hint object as possible. In case the hint is a container or a database, the new object will be created in the container or database.

HepODBMS contains clustering classes that allow clustering objects according to different algorithms. The **HepContainerHint** class is used to store objects in a series of containers or even databases, creating a logical container of unlimited size. Special iterators allow access to all of the objects later as if they were in one container.

5.2.3 Event Collections

LHC++ users will require both "normal" size and very large (10^9) event collections. HepODBMS provides the **h_seq<T>** class that presents the programmer with a single STL-like API for all types of collections. The actual implementation of the collection depends on a strategy object that can be supplied by a user. Currently implemented strategies include:

- Vector of object references
- Paged vector of references
- Single container
- Vector of container references

The **EventCollection** class is defined as below:

```
typedef h_seq<Event> EventCollection;
```

Figure 10 shows an example of how to iterate over a collection of events using an STL-like iterator.

```
EventCollection evtCol();              // Event collection
EventCollection::const_iterator it;    // STL like iterator

For( it = evtCol.begin();  it !=  evtCol.end(); ++it )
 Cout << "Event: " << (*it)->getEventNo() << endl;
```

Figure 10 Iterating over an event collection

6. CONCLUSION

HEP data stores based on Object Database Management Systems (ODBMS) provide a number of important advantages in comparison with traditional systems. The database approach provides the user with in a coherent logical view of complex HEP object models and allows a tight integration with multiple of OO languages such as C++ and Java.

The clear separation of logical and physical data model introduced by object databases allows for transparent support of physical clustering and re-clustering of data which is expected to be an important tool to optimise the overall system performance.

The ODBMS implementation of Objectivity/DB shows scaling up to multi-PB distributed data stores and provides integration with Mass Storage Systems. Already today a significant number of HEP experiments in or close to production have adopted an ODBMS based approach.

REFERENCES

[1] The Object Database Standard, ODMG 2.0, Edited by R.G.G.Cattell, ISBN 1-55860-463-4, Morgan Kaufmann, 1997

[2] C++ Object Databases, Programming with the ODMG Standard, David Jordan, Addison Wesley, ISBN 0-201-63488-0, 1997

BIBLIOGRAPHY

Object Data Management: Object Oriented and Extended Relational Database Systems R.G.G.Catell, Revised Edition, Addison-Wesley, ISBN 0-201-54748-1, 1994

The Object-Oriented Database System Manifesto M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pages 223-40, Kyoto, Japan, December 1989.

Object Databases As Data Stores for High Energy Physics, Proceedings of the CERN School of Computing, 1998, CERN 98-08

Object Oriented Databases in Hi High Energy Physics, Proceedings of the CERN School of Computing, 1997, CERN 97-08

The RD45 collaboration reports and papers, http://wwwinfo.cern.ch/asd/cernlib/rd45/reports.htm

RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1998, CERN/LHCC 98-x

Using an Object Database and Mass Storage System for Physics Production, March 1998, CERN/LHCC 98-x

RD45 - A Persistent Object Manager for HEP, LCB Status Report, March 1997, CERN/LHCC 97-6

Object Database Features and HEP Data Management, the RD45 collaboration

Using and Object Database and Mass Storage System for Physics Analysis, the RD45 collaboration

GLOSSARY:

ACID Transactions – Atomic, Consistent, Isolated, Durable
MB – Megabyte, 1 000 000 bytes
GB – Gigabyte, 1000 MB
PB – Petabyte, 1 000 000 GB
HEP – High Energy Physics
MSS – Mass Storage System
ODBMS – Object Database Management System
ODMG – Object Database Management Group (standards committee)
LHC - Large Hadron Collider
LHC++ - project aiming to replace the current CERNLIB libraries with a suite of OO software
DDL – Data Definition Language used by Objectivity/DB
OID – Object Identifier
MROW - multiple reader, one writer transaction where the old contents of a database region that is being modified by a writer is still accessible to other database clients in read-only mode

Object-Oriented Design of Minimization and Fitting Libraries in The Anaphe Project

Jakub T. Mościcki
IT/API, CERN 1211 Geneva 23, Switzerland
email: Jakub.Moscicki@cern.ch

Abstract

Minimization and fitting are key activities in physical data analysis. This paper presents design concepts of minimization and fitting software which is available as a part of Anaphe environment. In this context we focus mainly on software and computing issues. We discuss object-oriented approach to the design and implementation of numerical C++ class libraries and the use of Abstract Types (AIDA-compliant) as a way to achieve interoperability among various data analysis systems and across programming languages (C++, Java). Gemini and HepFitting are existing software components of Anaphe for minimization and fitting. We present the ongoing work on new class libraries.

1. Basic concepts of minimization and fitting

Data analysis in High Energy Physics (HEP) deals primarily with continuous, constrained minimization (Fig. 1), i.e., finding values of continuous variables that minimize given objective function while satisfying (optional) constraints. Typical data analysis task concerns fitting i.e. finding best set of parameters of a model function to represent experimental data using some fit criterion (like Least Squares, Poisson Maximum Likelihood [12]). In this context fitting is a disguised minimization problem where objective function is constructed in a special way and which usually has statistical interpretation. From a programming perspective this means that fitting may be implemented as a specific case of minimization and substantial part of code may be reused.

In the most general case nonlinear minimization problems may be very difficult to solve and human assistance may be required, both for the algorithm control (such as setting a starting point) as well as for the interpretation of the results (error and contour analysis). Particularly difficult, nonstandard problems require special treatment. However for a broad range of common problems there exist satisfactory, general algorithms. Fitting usually leads to well-posed and fairly easy minimization problems, so it is feasible to use general minimization algorithms to build fitting tools for HEP data analysis.

Minimization algorithms of our interest are usually iterative methods, which generate sequences of points that converge to the solution. Algorithms based on Newton or Quasi-Newton methods, such as Davidon-Fletcher-Powell Algorithm (DFP) [6], are commonly used in HEP analysis software (for example MIGRAD procedure of MINUIT Package [16]). These algorithms require computation of first derivatives in order to determine best search direction but usually computation of derivatives may be approximated numerically (for example by finite differences) and thus may be considered optional from the user point of view. Stochastic algorithms, such as simulated annealing ([14]) or linear problems solvers, such as the Simplex method ([21]), are usually less interesting in the particular context of HEP data analysis.

An introduction to minimization methods may be found in Numerical Recipes [23]. Further background information on optimization techniques may be found in [24].

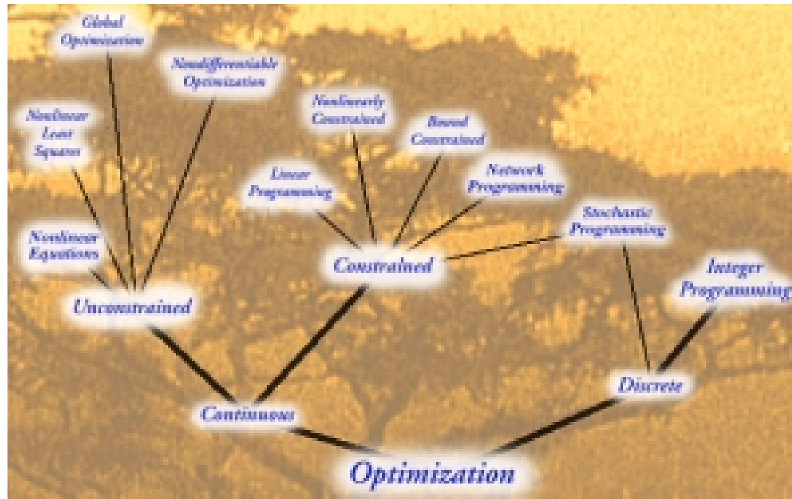


Fig. 1: Taxonomy of optimization problems (source: NEOS[20]).

2. Short introduction to The Anaphe Project

The Anaphe Project [2] aims to replace the current CERNLIB software libraries with a suite of OO software with roughly equivalent functionality. It provides a set of foundation, mathematical and graphical class libraries, visualisation toolkits and interactive analysis tools.

The layered approach promotes loose coupling between Anaphe components with clear separation between sampling and display. This allows to build relatively small, easily maintainable and well focused packages. Abstract Types agreed and defined in AIDA Project [1] help to clarify inter-package dependencies and to achieve interoperability between Anaphe and other HEP data analysis tools.

The Anaphe Project currently uses MINUIT FORTRAN Package [16] and Nag C Library [19] as minimization engines. MINUIT, which is a part of CERNLIB [4], has been widely used in HEP for more than 30 years. Nag C Library, a de facto standard in industry and research, contains high quality numerical algorithms including broad choice of minimizers. To provide C++ programmer with consistent, and customized for HEP-specific applications, way of interacting with the underlying minimization engines, Gemini [9] package has been created. Gemini provides a unified interface to both minimization engines and it also provides HEP-specific extensions if given minimization engine is missing them, such as Minos-type errors: " Special effort proved to be necessary in order to implement Minos-type error. Being standard within the HEP community, this type of errors is by no means standard in the non-HEP world. To the best of our knowledge, MINUIT was the only package which implemented this type of errors. We thus decided to write a special Minos analysis module for Gemini, so that any type of minimizer could be plugged-in, regardless on whether it is able to perform the Minos analysis or not". [26] In addition, there exists a small HepFitting [10] package which provides support for common fitting tasks.

The ongoing work aims to create new packages (libraries) with more flexible object structure and new interfaces. The rest of this paper is focused on the design of the forthcoming software.

3. The design of minimization and fitting packages

3.1 User interfaces for interactive analysis

Anaphe allows for interactive (commands and scripts) and batch-job (C++ programs) data analysis. For each kind of data analysis there exists a suitable programming interface which is implemented in an underlying library (Fig. 2). `IFitter` is a user interface, directly accessible from interactive and scripting environment (see Lizard Project [15]). It is simple to use and understand and allows for most typical analysis tasks. In general user interfaces like `IFitter` are by design *flat* and simple while interfaces

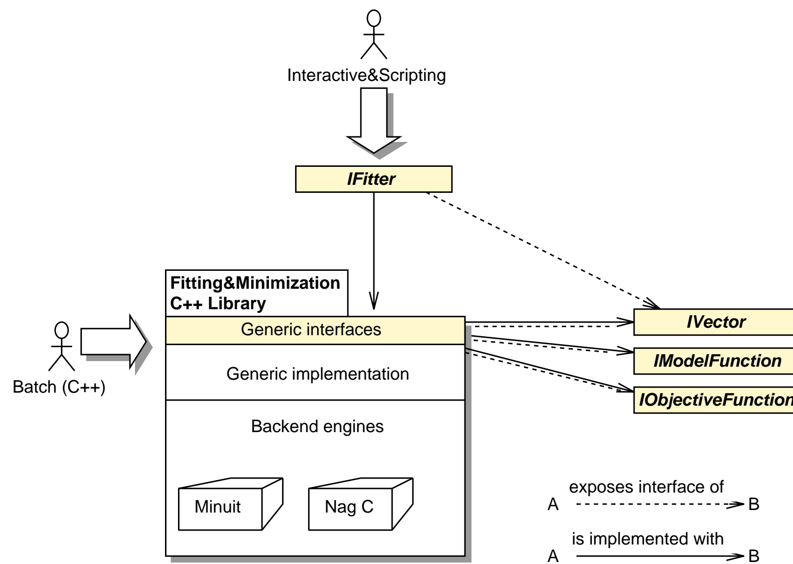


Fig. 2: The structure of Anaphe software for fitting and minimization at the level of packages and interfaces connecting them. Dashed arrows indicate coupling at interface level i.e. when one interface (IVector) forms a part of another interface (IFitter). Solid arrows indicate implementation.

of underlying library (available for C++ programming) are more complex but also more powerful. User interfaces can either be used directly by the end-user from a scripting language or indirectly, through GUI-based analysis system (see IGUANA Project [11]). Part of `IFitter` definition is presented below.

```
class IFitter  {
public:

  // Standard models (Gauss,Exp,Pol) described by string
  virtual void setModel(const char * )   = 0;
  virtual void setData(const IVector *)  = 0;
  virtual bool chiSquareFit()   = 0;

  virtual bool setParameter(const char *nameId, double start,
                            double step) = 0;

  // access parameters by a small IFitParameter interface
  virtual IFitParameter* fitParameter(const char *nameId) = 0;

  virtual bool includePoint(int i) = 0;
  virtual bool excludePoint(int i) = 0;

  virtual bool includeRange(int iMin, int iMax) = 0;
  virtual bool excludeRange(int iMin, int iMax) = 0;

  virtual void printParameters(ostream &os=cout) = 0;
  virtual void printResult(ostream &os=cout) = 0;
};
```

Default implementation for this interface is currently provided by Gemini and HepFitting but will be soon replaced by new libraries. `IFitter` comprises `IVector` interface to access data and, possibly in the future, `IModelFunction` interface to reference model function (if user defined models are needed). User interface for pure minimization has not been yet considered.

3.2 The Minimization Package

In this section we describe minimization software which may be used beyond particular context of fitting and thus is more general. Gemini is a current solution for HEP-specific Minimization Package in C++. It does excellent job to integrate underlying minimization engines and to provide missing features (Minos-errors) but does not offer a real object-oriented interface. New minimization C++ package consists of three parts. Generic abstract interfaces define generic functionality of minimizers and fitters for HEP data analysis. Generic implementation most of the time shadows generic interfaces, i.e. the class diagrams are very similar. Implementation details and specialities of back-end minimization engines are hidden. `IObjectiveFunction` interface is used directly to access objective function.

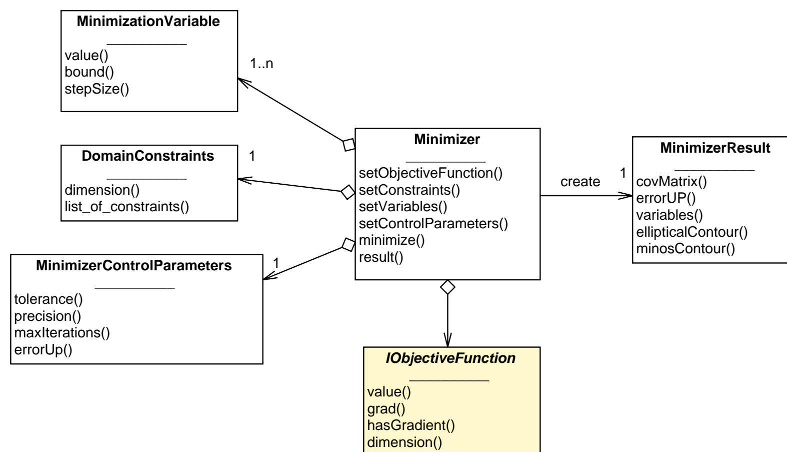


Fig. 3: Main minimizer objects: input objects, output object and objective function interface.

Generic `Minimizer` class uses several *input objects* for setting start-up configuration. Input objects include `DomainConstraints` to impose space constraints, a vector of `MinimizationVariable` to set staring point and bounds, and finally `MinimizerControlParameters` to set convergence criteria and to control error computation. `IObjectiveFunction` interface gives a way to access objective function. At the end of successful minimization, *output object* of class `MinimizerResult` is produced, which stores the results. With such a design it is easy to maintain clear state definition of the minimizer. It is also easy to reproduce minimization because the minimizer state may be captured and externalized in input and output objects. `Minimizer` does object management in such a way that input and output objects which are accessible from minimizer are always in sync.

While `Minimizer` provides generic interface, specific minimizers are implemented by subclassing (Fig. 4). `Minimizer` also defines the life-cycle of the underlying engines with the help of abstract, protected *factory methods*. This allows to overcome some deficiencies of underlying engines. Common problem concerns MINUIT global variables, which make parallel use of several minimizer objects in one program impossible. With the proposed solution interferences between multiple MINUIT-based minimizers are eliminated. It must be emphasized that this technique does not address the issues of thread-safety.

Strategy “specific minimizer by subclassing”, which has been already used in Gemini, allows to use generic minimization interface and to switch between different engines without affecting client’s code. This may also be helpful in transition period for the code relying on Fortran libraries.

Objective function is accessed via `IObjectiveFunction` interface, which in the future will be agreed across different analysis tools. The actual implementation of the function may be done in context of other languages (like Java) or analysis frameworks. It also enhances flexibility because objective function may be implemented within CORBA [5] infrastructure and computation may actually take place

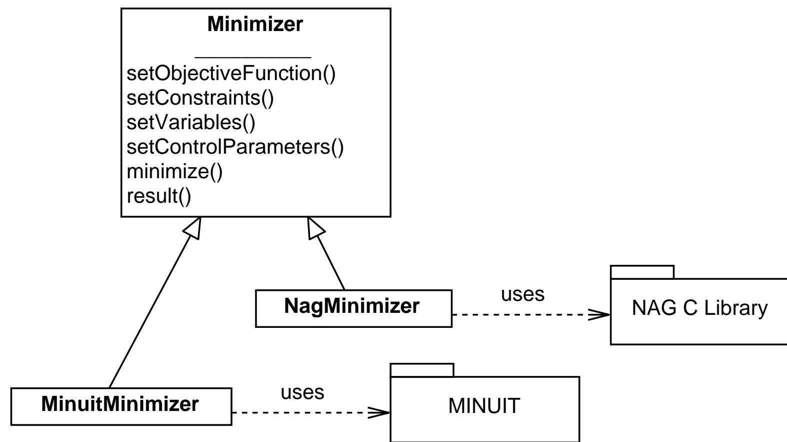


Fig. 4: Specific minimizer are subclassed from a generic class.

on a distant machine.

Simple example of using minimizer in C++ program, minimizes the Powell's quartic function of four variables subject to the constraints: $x^2 + y^2 + z^2 + u^2 = 1$ which defines a 4D sphere which is intersected with a 4D plane $x + y + z + u = 0$.

```
int main()
{
  PowellQuarticFunction objfun;
  ConstraintForPowell ctrfun;

  NagMinimizer nlp(&objfun);
  DomainConstraints dcs(4);

  vector<MinimizationVariable> mvars(4);

  dcs.set( NonlinearConstraint(&ctrfun, 1.0, 1.0) );

  // all coeffs of the linear equation are equal 1.0
  dcs.set( LinearConstraint(vector<double>(4, 1.0), 0.0, 0.0) );

  nlp.setConstraints(dcs);

  // take random initial values between -3 and 3
  // (the minimizer will project it onto the admissible set)
  for(int i=0;i<4;i++)
    mvars[i] = MinimizationVariable(-3.0 + ((float)rand()*6)/RAND_MAX);

  nlp.setVariables(mvars);
  nlp.printSetup(cout);
  if( nlp.minimize() == false) exit(1);
  nlp.printResults(cout);
}
```

Program above uses function objects for objective function and constraint function. `PowellQuarticFunction` implements `IObjectiveFunction` interface:

```
class PowellQuarticFunction : virtual public IObjectiveFunction
{
```

```
 public:

  double value(const vector<double>& parms) const
  {
    const double &x=parms[0], &y=parms[1], &z=parms[2], &u=parms[3];

    return square(x+10*y) + 5*square(z-u) + square(square(y-2*z))
      + 10*square(square(x-u));
  }

  const vector<double>& grad(const vector<double>& parms) const
  {
    const double &x=parms[0], &y=parms[1], &z=parms[2], &u=parms[3];

    m_grad_buf[0] = 2*(x+10*y) + 40*(x-u)*square(x-u);
    m_grad_buf[1] = 20*(x+10*y) + 4*(y-2*z)*square(y-2*z);
    m_grad_buf[2] = 10*(z-u) - 8*(y-2*z)*square(y-2*z);
    m_grad_buf[3] = -10*(z-u) - 40*(x-u)*square(x-u);

    return m_grad_buf;
  }

  bool hasGradient() const { return true; }
  int dimension() const { return 4; }

  PowellQuarticFunction() : m_grad_buf(4) {}

  private:
  vector<double> m_grad_buf;
};
```

`ConstraintForPowell` implements `IFunction` and its interface is very similar to the `IObjectiveFunction`:

```
class ConstraintForPowell : public IFunction
{
 public:

  double value(const vector<double>& x) const
  {
    return square(x[0])+square(x[1])+square(x[2])+square(x[3]);
  }

  const vector<double> grad(const vector<double>& x) const
  {
   for(int i=0; i<dimension(); i++)
    m_grad_buf[i] = 2*x[i];
   return m_grad_buf;
  }

  int dimension() const { return 4; }
  bool hasGradient() const { return true; }

  ConstraintForPowell() : m_grad_buf(4) {}

  private:
```

```
  vector<double> m_grad_buf;
};
```

3.3 The Fitting Package

As it was already mentioned, fitting is just a special case of minimization. Given a data set, model function and the criterion to estimate "goodness" of the fit (fit method), suitable objective function may be constructed. The rest of the job may be done by a minimization software. As shown in Fig. 5, almost all fitting-specific work is done at the function level. Once the appropriate objective function is constructed, `Fitter` uses an instance of `Minimizer` to perform minimization.

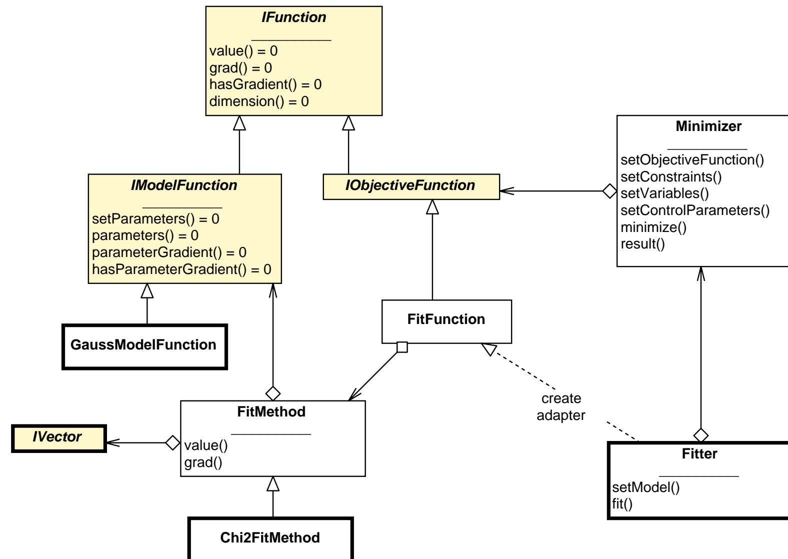


Fig. 5: Fitter design. User typically deals with classes marked with bold box outline.

Interface `IModelFunction` defines function with setable parameters. Implementation of common model functions are provided: `GaussModelFunction`, `ExponentialModelFunction` and `PolynomialModelFunction`. Any algebraic combination of these may be used or user may provide hand-written models. Model function is used by `FitMethod`, which is an abstraction of fit method algorithm. Two standard algorithms are provided: `Chi2FitMethod` (least squares) and `LogMLFitMethod` (Poisson maximum likelihood, especially useful with small data sets). The main idea behind this design is to separate development of model functions and fitting methods - each piece may be developed in an independent manner. `FitFunction` combines these pieces together and connects `FitMethod` with `IObjectiveFunction` through an *adapter* pattern [8]. This step is hidden from user and it is performed automatically by the `Fitter`. Data set is accessed in a storage-independent manner via `IVector`. It may be read from a variety of places such as plain ASCII file, XML stream or object database (ODBMS [22]). Apparent complexity of this design is largely reduced by the fact that typically user deals only with specific instances of model functions and fit methods (these classes are marked with bold outline in Fig 5).

Simple example of fitting is presented below. `AIDAVector` is a default Lizard [15] implementation of `IVector`. `NagEngine` object is used to indicate the type of the minimization engine and `Chi2Fit` to indicate the type of fitting method.

```
main()
{
  Fitter fitter(NagEngine);
```

```
  AIDAVector v;

  v.fromAscii("testing-1d");

  GaussModelFunction f(1,1,0);

  fitter.setModel(Chi2Fit,&v,&f);

  if(fitter.fit())
    {
      cout << "fitting successful!";
      fitter.minimizer()->printResults();
    }
  else
    cout << "fitting failed!";
}
```

4. Concluding remarks

Anaphe packages for minimization and fitting are designed to offer maximum flexibility to users and developers. Through the layer of object interfaces they offer access to powerful minimization engines. The implementations are 100% compliant with C++ Standard [3] and make use of Standard C++ Library [13], in particular of STL [18]. Design pattern [8] are applied wherever suitable. Unified Modeling Language (UML) [7] is used for the analysis and design.

The intent is to provide the packages for general HEP data analysis. Therefore it must be stressed that fitting and minimization software in Anaphe, similarly to CERNLIB minimization packages, is not intended "for repeated solution of identically parametrized problems (such as track fitting in a particle detector) where a specialized program will in general be much more efficient" [16].

The testing is based on standard and verified methods. Minimizers are tested using Moré set of test functions [17]. Statistical Reference Datasets [25] will be used to validate upcoming version of fitting package.

User interfaces allow to perform common tasks within the framework of Lizard [15] analysis environment, but without overloading the user with the whole complexity of underlying class library (C++ package). The class library may be fully accessed from the C++ programs.

References

[1] AIDA: Abstract Interfaces for Data Analysis, http://aida.freehep.org/

[2] Anaphe: Libraries for HEP Computing, http://wwwinfo.cern.ch/asd/lhc++

[3] ANSI/ISO C++ Standard. http://webstore.ansi.org/AnsiDocStore

[4] CERN Program Library, http://wwwinfo.cern.ch/asd/cernlib

[5] CORBA: Common Object Request Broker Architecture, http://www.omg.org

[6] Fletcher R., Powell M.J.D.: A rapidly convergent descent method for minimization, Computer Journal, Vol.6, 1963

[7] Fowler M., Scott K., Booch G.: UML Distilled: A Brief Guide to the Standard Object Modeling Language (2ed ed), Addison-Wesley Pub Co; ISBN: 020165783X

[8] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, ISBN 0-201-63361-2

[9] Gemini: A minimization and error analysis package in C++, Reference guide: http://wwwinfo.cern.ch/asd/lhc++/Gemini

[10] HepFitting: A C++ fitting API for HEP based on Gemini, Reference guide: http://wwwinfo.cern.ch/asd/lhc++/HepFitting/

[11] Iguana: Interactive Graphical User Analysis, http://iguana.cern.ch

[12] James. F.E et al: Statistical Methods In Experimental Physics, ISBN 0 7204 0239 5, North-Holland Publishing Co, 1971

[13] Josuttis N.M.: The C++ Standard Library : A Tutorial and Reference, Addison-Wesley Pub Co; ISBN: 0201379260

[14] Kirkpatrick S., Gelatt C.D., Vecchi M.P.: Optimization by Simulated Annealing, Science 220(4598): 671-680(1983)

[15] Lizard: an AIDA compliant Interactive Analysis Environment, http://wwwinfo.cern.ch/asd/lhc++/Lizard/

[16] MINUIT - Function Minimization and Error Analysis, Reference Manual CERN Program Library Long Writeup D506, also http://wwwinfo.cern.ch/asdoc/minuit/minmain.html

[17] Moré J.J., Garbow B.S., Hillstrom K.E.: Testing Unconstrained Optimization Software, ACM Trans. Math. Software 7 (1981), 17-41

[18] Musser, D.R., Saini, A.: STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Addison Wesley, ISBN 0-201-63398-1.

[19] Nag C Library, http://www.nag.co.uk/numeric/cl/CLdescription.asp

[20] NEOS Guide: Network Enabled Optimization System Guide, http://www-fp.mcs.anl.gov/otc/Guide

[21] Nelder J.A, Mead R.: A simplex method for function minimization, Computer Journal 7 (1965), 308-313.

[22] Object Database Management Group: The Object Data Standard: ODMG 3.0. Morgan Kaufmann Publishers, ISBN: 1558606475, also http://www.odmg.com

[23] Press W.H., Teukolsky S.A., Vetterling W. T., Flannery B. P.: Numerical Recipes in C, The Art of Scientific Computing, 2nd Edition, Cambridge University Press; ISBN: 0521431085, also http://www.nr.com

[24] Russenschuck S.: Mathematical Optimization Techniques, In: 1st International Roxie Users Meeting and Workshop, CERN, Geneva, Switzerland, 16 - 18 Mar 1998

[25] Statistical Reference Datasets, http://www.nist.gov/itl/div898/strd/

[26] Szkutnik Z.: Gemini and HepFitting components of LHC++, In proc. 1999 CERN School Of Computing, also: http://wwwinfo.cern.ch/asd/lhc++/TUTORIALS/index.html#CSC99

STORAGE AND SOFTWARE FOR DATA ANALYSIS

R. Brun[1], R.G.Jacobsen[2], R.Jones[1], J. Moscicki[1], M.Nowak[1], A.Pfeiffer[1], F.Rademakers[3]

1) CERN, Geneva, Switzerland
2) University of California, Berkeley, USA
3) GSI, Darmstadt, Germany

Abstract

This track combined exposure to the software technologies and packages relevant for LHC experiments and the engineering aspects of software development. The lectures provided an overview of LHC++/Anaphe and ROOT and covered those aspects of software engineering most relevant for HEP software development. It showed, in a practical sense, how software engineering can help in the development of HEP applications based on the LHC++/Anaphe and ROOT software suites and also gave a taste of working on large software projects that are typical of LHC experiments. A series of hands-on tutorials performed by the students were based exercises to solve given problems. The tutorials followed the natural progression of physics analysis exploring the major packages of LHC++/Anaphe and ROOT on the way.

This paper presents an overview of the software engineering lectures by R.Jones, the tutorials and feedback session. Details of the lectures on LHC++/Anaphe and ROOT are reported in separate papers of these proceedings.

1. INTRODUCTION TO SOFTWARE ENGINEERING

A definition of what is meant by software engineering gave a starting point for this lecture which then went on to explain how the scale of the software project determines the software process required to successfully run the project to completion. Developing a model for a large scale software system prior to its construction or extension is as essential as having a blue-print for constructing a building. Good models are necessary for communication between the project stakeholders (members of the development team, users and management) and to assure architectural soundness. As the complexity of the software system under development increases so does the importance of good modelling techniques.

Various processes exist for object-oritened (OO) software (OOIE, OMT, Booch, Fusion, Syntropy, OOSE, Unified etc.) and have varying definitions for the phases involved during the project. The history of these OO software development proceses was described and how this lead to the appearance of the Unified Software Development Process (USDP) as a defacto market standard taking elements from previous development processes. The authors of USDP recognised the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. USDP is centred around the architecture of the software under development and follows a number of iterations driven by use-cases. Each iteration can be treated as a mini-project resulting in a new release of the software and following all the phases of the software process.

The importance of a notation to document and visualize the models developed as artifacts of the process phases was explored and the structure of the Unified Modelling Language (UML) notation

was shown. The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems. USDP incorporates the UML as a standard into its development processes and products, which cover disciplines such as requirements management, analysis & design, programming and testing. The UML can be used on varying software development projects since it is independent of the process (i.e. USDP) being followed. An overview of the UML starting with use-cases showed the basic structure of the UML, the notation and types of diagrams that can be used to describe the software under development. Emphasis was put on use cases as a means of driving the development and how they are used to capture the requirements. The essential purpose of the requirements gathering is to aim development towards the right system. This is achieved by describing the requirements (i.e. the conditions or capabilities to which the system must conform) well enough so that an agreement can be reached between the stakeholders on what the system should and should not do. The issues involved in ranking use-cases to determine their priority and establishing how they map onto the project iterations was described.

Once the use-cases have been explored and ranked attention can move onto establishing the basic architecture of the software starting with deployment diagrams showing the nodes (processors) and interconnections (networks) that represent the environment in which the software will run. This environment is populated with the various modules of the software identifying interfaces to legacy systems (hardware and software) and the relationships between components. Details of the software are omitted from such diagrams to allow the basic disposition and architecture to appear.

2. SOFTWARE DESIGN

This lecture carried on the description of architecture and showed how work progresses towards establishing a more detailed design. The task of design was introduced as consisting of three levels: architecture, mechanistic and detailed based on the scope of the decisions made. Each level was further defined to show it's goals, techniques and deliverables. At the archiecture level the emphasis is on identifying the major software structures such as subsystems, packages and their interconnections. At this point it is possible divide the work into smaller tasks based on domains or subsystems that can be developed by different teams or individuals possibly in parallel. The desirable qualities of archiectural design was listed including layering sub-sytems to reduce coupling and promote independence. Other qualities such as well-designed interfaces and scalability were also mentioned but the most important quality of any design is that it is easily understandable.

At the mechanistic level attention moves to establishing the relationships between groups of classes (i.e. the mechanisms by which they are linked) and at the detailed level the developer examines the internal structure of individual classes by identifying necessary attributes and methods. The transition to implementation is made selecting the appropriate mapping for aspects such as associations and operations taking into account deployment (e.g. the association between two classes may be different if they are on separate machines or in separate threads), code ownership and practicalities linked to the use of underlying software packages. The UML class, sequence and collaboration diagrams were explained and examples drawn from the exercises. As design progresses through the three levels, more detail is added to existing diagrams (e.g. adding methods and attributes to classes) and new ones are drawn.

The concept of patterns that give examples of how several classes can work together in a given domain to address some problems was introduced. The ability to profit from design approaches used in other projects was discussed as a means of supplimenting the developers' knowledge and experience. Examples of how to apply patterns to analysis and design taken from LHC++/Anaphe and ROOT were given. While patterns are rightly considered as an excellent aid during analysis and design, the audience were warned about their abuse and some of their short-comings.

Finally an appreciation of the advantages and problems of UML were described as well as future likely changes proposed for inclusion in the release 2.0 of the standard scheduled for 2001.

3. SOFTWARE TESTING

This lecture covered the basic principles of software testing and why programs have defects.

The goal of software testing is to ensure the software under development has sufficient quality by evaluating the artifacts (documents, diagrams, source code etc.) of a project. The intention is to find defects (bugs) so that they can be removed and demonstrate that the software does meet its specifications. As a side effect, it builds confidence in the project members that the software is ready for use.

Testing includes both validation and verification of software. Verification implies executing an implementation of the software and has traditionally be the focus of software testing in HEP. This implies that no software testing is performed before an implementation is ready (i.e. once the requirements, design and implementation phases have been completed). A complimentary activity is that of validating the artifacts at each stage of the software process. For example, the design diagrams can be validated to ensure they satisfy all the requirements. The requirements document can be validated to ensure it is consisitent, complete and feasible.

Once an implementation exists, it can be validated against the design diagrams to ensure it faithfully realises the design and exhibits general design qualities (listed above). Validation has been shown to be more effficient that verification because each artifact of the software process is checked as it is produced and so errors are caught earlier when they cost less to correct.

The different phases of software validation and verification were enumerated and described:

Unit tests – tests peformed on individual classes;

Integration – tests performed on several classes, components or sub-systems to validate their interfaces;

Regression – repeating previously executed tests after a modification has been made to ensure the defect has been removed and no new ones have been added;

Acceptance – final testing performed at the user's site with their data.

Different techniques can be applied for validation and verification. Reviews and inspections are the most appropriate manner of validating documents, diagrams and source code. Three types of reviews were listed:

A personal review where an individual developer examines their own artifact individually;

A work-through where a developer presents their artifact to her/his fellow developers (peers) who are asked to comment and make suggestions;

An inspection which is a structured review involving an inspection leader, the author(s) and a number of peer reviewers. A number of meeting are held to inspect the artifact and all issues are logged and followed-up by the authors(s).

A number of elements need to be put in place before a review can take place:

A checklist of the most likely errors must be available to drive the process and indicate to the reviewers what they should be looking for;

The project stakeholders must accept that a review will "front-load" the cost of a project since the review will take some effort (even if it is only a couple of hours) and that the development

cannot proceed to the next phase until it is completed while (hopefully) the total amount of testing required will be reduced;

The review should be seen as a phase of the software process and not a personal assessment of the authors;

Training will be required for inspection leaders if the process is to run smoothly to a successful completion in a reasonable amount of time.

As an example, guidelines were shown for inspecting C++ sources code. Much of the work involved in inspecting source code can be automated using a coding rule checking tool to parses source code files and produce an evaluation report identifying which guidelines were violated. This report can then used to make modifications to the source code where necessary but violations of the guidelines should be accepted where they can be justified.

Techniques for validation were also addressed including black-box and white-box testing. In black-box testing the specifications and interfaces of the software is validated without examination of the structure of the source code itself. While in white-box testing, knowledge of the internal structure of the software is used to develop suitable test-cases. Boundary-value testing (e.g. varying the input parameters to object methods) was given as an example of black-box testing.

Coverage and path testing where knowledge of the internal structure of the software is used to develop test-cases that execute as many paths through the code as possible were given as examples of white-box testing.

Again, CASE tools can help to automate important fractions of the work involved in black and white box testing. The example of the Insure++ tool which can be used for code coverage of C code was given. Similarly tools exist for static source code analysis (e.g. LINT, Logiscope), memory leak checkers (e.g. Purify, Insure++) and boundary checks (e.g. "T" testing tool). Performance of the software and identification of bottlenecks can be identified by compiler profilers and scripting languages such as Expect are very useful for writing tests-cases.

Following the assumption that prevention is better than cure, a number of programming techniques available in many high-level languages were listed as being particularly error prone and difficult to master. The list includes dynamic memory allocation, parallelism, recursion, floating point numbers, interrupts and pointers. Unfortunately, these programming techniques are amongst those that improve the performance or efficiency of software and since HEP software is often at the limit of what is possible many of these techniques and often necessary.

The lecture concluded with a set of axioms about testing that can improve the way most software developers approach the subject and by trying to answer the most difficult questions concerning software testing:

How much testing is enough?

When or why should we stop testing?

When is the software ready to be released?

Since no software can be fully tested the concept of risk-based testing as a means of prioritising the choice of test-cases to be made was suggested. With such an approach, test-cases are developed to verify that the most important risks have been addressed while testing on low-priority aspects of the software may be dropped based on general agreement between stakeholders of the project.

4. LONG-TERM ISSUES OF SOFTWARE

This aim of this lecture was to look at some aspects which affect the long-term well-being of development projects. Once the software has been developed the emphasis moves to its maintenance. The cost of software maintenance usually exceeds the cost of software development. There are three principal types of software maintenance:

perfective - where new functionality is added to the system;

adaptive - where the system is adapted to new enviornments (e.g. ported to a new operating system);

corrective - which is removing defects from the software.

Having defined software maintenance, the lecture moved on to look at how to minimise its cost and assure successful completion of the project by asking three questions:

Why is the software process so important?

What is so good about iteravtive development anyway?

Why can't we just get on with writing the code?

To answer the first question, the most common reasons for failure of software projects were listed. An analogy based on building bridges showed how activities, such as adequate analysis and design, can be used to avoid them. The second question was addressed by giving an example of what iterative development means and by showing the unfortunate results of not using it.

Hopefully the students understood that by answering the first two questions the answer to the third becomes clear. As a means of supporting interative development cycles, configuration management systems were introduced and the lecture finished by emphasising that software always costs something (time or money): either *some* up-front by investing in analysis and design or *more* later to fix all the problems.

5. EXERCISES

The hands-on tutorials included a series of exercises to solve given problems. The tutorials followed the natural progression of physics analysis exploring the major packages of LHC++/Anaphe and ROOT on the way. The students completed the tutorials in groups of two. The students were required to develop several C++ programs in succession starting from skeletons:

1. Generate a set of events to be stored on disk according to a defined object model thereby exploring the issues of data persistency;
2. Build a set of event tags (Ntuples) from data prepared in 1 and identify/calculate interesting event attributes;
3. Use the minimization packages to find the minima values for a given set of problems
4. Read event tags built in 2 and display the contents. Use the interactive graphical tools to apply more cuts.

The LHC++/Anaphe and ROOT lectures are documented as separate papers in these proceedings. Below is a summary of the software engineering lectures by Bob Jones and the feedback session.

6. FEEDBACK SESSION

The track finished with a feedback session during which answers were given to the questions asked by the students about different aspects of the software suites covered by the track (ROOT and LHC++). The JAS software suite, though officially part of another track of the school, was included. The

students had submitted written questions to which the developers of each software suite had provided written answers. These answers were collected together and put on the school web pages. The major issues covered by the questions were:

Data storage

Interfacing to external code, experimental packages

Scaling

How does this work relate to GRID?

A subset of the questions were presented during the feedback session for further discussion. The answers provided by the authors of each software suite were shown and then follow-up questions were asked by the students. The questions were as follows:

OBJECTIVES AND APPROACH

ROOT

With the ROOT system, written in C++, we provide, among others, an efficient hierarchical object store, a C++ interpreter, advanced statistical analysis (multi dimensional histogramming, fitting and minimization algorithms) and visualization tools.

The user interacts with ROOT via a graphical user interface, the command line or batch scripts. The command and scripting language is C++ (using the interpreter) and large scripts can be compiled and dynamically linked in.

ROOT also contains a C++ to HTML documentation generation system using the interpreter's dictionaries (the reference manual on the web is generated that way) and a rich set of inter-process communication classes (supporting TCP/IP and shared memory). For the analysis of very large datasets (> TB's) we provide the Parallel ROOT Facility (PROOF).

The system is packaged in a set of modules (shared libraries) which are dynamically loaded only when needed.

The ROOT project was started in January 1995 to provide a PAW replacement in the C++/OO world. The first pre-release was in November 1995 (version 0.5) and the first public release in fall 1996.

LHC++/Anaphe

The Anaphe/LHC++ project aims at replacing the full suite of functionality formerly provided by CERNLIB. Amongst those packages the analysis tool (Lizard) is one component. It uses a set of fundamental libraries (like, e.g., HTL, CLHEP, HepODBMS) which have been developed in the last couple of years in close collaboration with experiments (mainly LHC but also other CERN and non-CERN experiments) and other HEP projects (such as Geant-4). We try to make use of good software engineering practices to improve the quality and long-term maintainability (UML, use cases, tools etc.)

The aim is to provide a flexible, interoperable, customizable set of interfaces, libraries and tools. Re-use of existing (public domain or commercial) packages as far as possible. Writing HEP specific adaptations wherever needed. Taking into consideration the huge data volume expected for LHC, the distributed computing necessary to analyse the data as well as long-term evolution and maintenance.

The use of an OO DB with transaction safety (locks) guarantees consistency in the datasets written. This is especially important in a distributed/concurrent environment. The basic

libraries exist since about 1997. Development of some parts (AIDA, Plotter, and Lizard) started in fall 1999. The first release is scheduled for October 2000.

JAS

Leverage the power of Java as much as possible because:

Provides many of the facilities we need as standard.

Is easy to learn and well matched (in terms of complexity) to physics analysis

Is a mainstream language, so time spent learning it is well spent?

Is a high performance language (see Tony's talk)

Is a highly productive language (no time wasted debugging core dumps).

JAS has been in development for 4 years (since Hepvis 96)

HOW DOES THE SOFTWARE WORK WITH NON-NATIVE DATA STORAGE?

If an experiment defines its own storage system, can the software suite use it? What capabilities are lost in that case? Specifically, can ROOT/JAS work with HepODBMS/Objectivity without losing capability? Can JAS/LHC++ work with ROOT files without losing capability?

JAS

JAS does not have a "native" data format, it can work with any data format for which a DIM exists. DIM's already exist for PAW, ROOT and Objectivity and many other formats; it is fairly easy to create new DIMs for experiment specific data.

The more detailed question is harder to answer, the specifics depend mainly on how completely the DIM has been implemented. For example the current Objectivity DIM is only able to read HEPTuple data from Objectivity databases. Objectivity does have a Java binding, so writing a more fully functioned interface is possible, although there are some complications arising when attempting to read data initially stored into Objectivity from C++, especially if no thought was given to Java access up front.

ROOT

Root can read any type of data not in Root format. The typical situation is to read ASCII files or any type of binary data via normal C++.

The h2root program is an example of a C++/Root based program converting PAW files to Root format.

NASA has implemented an interface between Root and the HDF files that are the standard for AstroPhysics.

Root has interfaces to RDBMS systems such as MySQL and Oracle. An ODBC/JDBC interface has been developed by Valery Onuchyin (see link on Root web site). We have tens of examples of collaborations using their legacy data and processing them with Root.

LHC++/Anaphe

In Lizard you have access to all the experiment's code and data in their native (storage) format using the Analyzer. If you want to store the histograms/ntuples using their own storage system, an implementation (adapter) of the Histo/Ntuple Abstract Interfaces is needed.

Questions from the audience:

Q: Can you use ROOT and Objectivity together?

A: (Rene) There are some implementations already developed.

Q: It would be useful to have a standard tool to convert ROOT files to Objectivity...

A: (Rene) We haven't receive any request for this.

Q: How is the distribution of data made in BaBar (is it Objectivity)?

A: (Bob Jacobsen) We use both Objectivity and ROOT. We have observed the same performance within 10%. We have 8 large sites using Objectivity and many small sites using ROOT.

Q: Could someone give me a definition of Objectivity?

A: (Andreas) It is a commercial product which enables the user to provide OO schemas to describe his/her databases. It describes and stores his/her data model in a completely

transparent (to the program) and location independent way. It provides all the regular database features.

LHC DATA SIZES

What will need to be developed to handle the expected size of LHC data analysis? What are the current strengths and weaknesses for storing very large amounts of data?

JAS

The strengths of JAS are in its ability to adapt to whatever data format is eventually decided upon, and to support access to very large datasets using its distributed client-server mode. There are some weaknesses in the current java.io package when dealing with large amounts of binary data, but these will be addressed by the addition of a new java.nio package in the next release of Java (JDK 1.4 scheduled for release next summer). After which there is no reason to expect Java IO will be any less efficient than C++ IO.

ROOT

Root is assumed to work in conjunction with an RDBMS. The RDBMS handles the run/file catalog and other data that require locking, journaling, etc. Root files have a current practical limitation to 2 GBytes. All the hooks are already in Root to support larger file sizes. The Alice data Challenge has demonstrated the storage of 25 TeraBytes of data with the run catalog (25000 files) stored in a simple MySQL database. A run catalog of 1000000 files has been successfully tested with MySQL. Many experiments are currently experimenting with the combination Root + RDBMS and expect to store several hundred TeraBytes in 2001.

LHC++/Anaphe

In LHC++, there were a number of studies done on how Objectivity/DB scales to store several petabytes of data. Presently, only the fixed organization of the Object-Identifier limits the amount of data that can be stored in "small" (few GB) files per Federation.

During the last year, the BaBar experiment has accumulated more than 100 TByte of data in Objectivity/DB thereby showing scaling behaviour to amounts of data that are only one order of magnitude lower than the ones expected for LHC.

As stated above, the use of an OO DataBase with transaction safety (locking) guarantees consistency in the datasets written. This is especially important in a distributed/concurrent environment.

Questions from the audience:

Q: Is it possible to store files with sizes above 2Gb?
A: (Bob Jacobsen) I don't want too many files but I don't want very big files either.
A: (Rene) ROOT has a current limitation to files of 2 GB. This is the limitation imposed in general by the OS. All the hooks are in the system to support larger files in a backward compatible way. You need different separate techniques to store run catalog and events store.
Q: How can we retrieve huge amounts of data rapidly?
A: (Bob Jacobsen) None of these software suites has addressed all the questions, but we know now what the questions are.
Q: How is Objectivity used in BaBar?

A: (Bob Jacobsen) It is used as an object store, but there is something on top to find where the event is, so that access to data is location-independent. We created an API on top of Objectivity to provide this functionality, which is needed by every experiment.
A: (Rene) A remark on performance. In ALICE simulation Objectivity and ROOT differ by a factor 5 in size. There is a factor from 3 to 5 in real time for accessing data (ROOT being the faster).
A: (Bob Jacobsen) I have a comment on benchmarks done by people who know their own system very well and learned the other one in a short time... However Babar created its own way of retrieving data.

COMPATIBILITY WITH EXTERNAL SOFTWARE

How does the software work with external software such as GEANT4 and GEANT3? What can you do and not do with them?

JAS

We demonstrated the use of JAS with Geant4 during this school. The Geant4 collaboration is considering a proposal to adopt the AIDA interface as a standard interface to histogramming in Geant4, meaning that it will be easy for Geant4 to interact with any AIDA compliant analysis tool.

We have interfaces with Root, Objectivity, PAW, WIRED, G4, StdHEP, and AIDA. Due to the simple "plugin" mechanism we expect to develop many more.

ROOT

The Alice collaboration has developed an abstract MC interface in the AliRoot framework. This abstract interface has currently an implementation (TGeant3) for Geant3 and a set of classes. TGeant3 is independent of Alice

- AliGeant4 with Alice specific classes for Geant4
- TGeant4: an experiment independent interface to G4 called by AliGeant4

The geometry input and the hits output is MC independent (same classes for Geant3 and Geant4).

A Root GUI (for GEANT4) has recently been developed in Alice by Isidro Gonzalez. This work will be presented at the coming Geant4 workshop in October.

Also in Alice, Ivana Hrivnakova who developed the AliGeant4 and TGeant4 classes is currently working in processing all the G4 classes via rootcint. This work is now close to completion. Once it is done, the following facilities will be available:

- Automatic I/O of G4 objects
- Automatic inspection and browsing of G4 data structures
- Cint interactive interface to the G4 classes.

LHC++/Anaphe

In Lizard access to Geant-4 (as to any other external software) is through the Analyzer. In addition, Geant-4 is developing AIDA based Abstract Interfaces to be used in connection with analysis packages.

As for the experiments, in the context of LHC++ we have been working closely together with Geant-4 for the design/implementation of the object persistency using an Object Database.

Questions from the audience:

Q: Can LHC++ interface GEANT3?
A: (Andreas) This can be done through the package. Be aware of limitations of using Fortran code (especially with common blocks) in multi-threaded environments which are becoming more and more popular.
A: (Rene) GEANT3 and 4 can be interfaced in the same way in ROOT. The idea to develop an abstract interface in ALIROOT was to easy the transition from GEANT3 to GEANT4. Thanks to this interface, Alice had the same geometry input and the same output objects with G3 and G4.

EXISTING EXPERIMENTAL SOFTWARE

If an experiment has an existing software package how do you interface it and how much of its capability will be available?

JAS

In principle, using a combination of plugins and DIM's you should be able to interface any experiment to JAS. In practice in depends how "Java Friendly" the experiment is (extensive use of C++ features such as templates tend to make it more difficult). Well-designed, modular experiment software also helps. The person who builds the JAS interface will need to learn a fair bit about Java and JAS, but once that is done it should be easy for other collaborators to use the interface.

Direct interface with C++ code is currently a weak point of Java, thus direct interface with C++ experiment code is currently difficult. We expect more and more experiments to adopt Java as the huge productivity benefits of using Java become more widely appreciated, meanwhile we are attempting to address this issue via the development of tools such as JACO, and plan to test this in the context of the Atlas event model. Interfacing with experiment software in Java (such as LCD) or via some intermediate storage format (e.g. PAW, Objectivity, ROOT) is comparatively straightforward.

ROOT

There are two ways to use ROOT. As a framework or as toolkit. When using it as a framework ROOT will control the event loop and call your software. This will allow you to use all ROOT's capabilities. Or you can use ROOT as a toolkit you keep your event loop and you use ROOT features like a 'sub-routine library' calling e.g. ROOT's histogramming, I/O, etc. Both approaches are used by several experiments. ALICE, STAR, etc. use the first, while ATLAS, LHCb are using the latter.

LHC++/Anaphe

If the experiment's s/w package is independent of the categories used in Lizard, its full capability is available through the Analyzer. In case the experiment wants to replace one or more of the categories, one or more Adapter to the Abstract Interface needs to be written (if not already done).

Questions from the audience:

Q: (Bob Jones) How is it possible to interface the suites with other software components (e.g. CORBA)?
A: (Andreas) We will address that problem next year.

A: (Bob Jacobsen) It has been done in Babar. We use JAS via CORBA (it is simple to interface JAVA with CORBA).

A: (Rene) There is no problem, since we already developed socket-based high performance communication. We could implement CORBA as well.

IF I WANT TO MAKE AN IMPROVEMENT, HOW DO I GO ABOUT IT?

LHC++/Anaphe

For the HEP part of Anaphe/LHC++ download the sources from the CVS repository (or the tgz file), modify and let us know!

ROOT

Proceed like several thousand people who are already doing this. Subscribe to the roottalk mailing list, see what happens and feel free to submit your complaints, additions, etc. The Root web site has a long list of links to users' contributions. The CREDITS file in the Root source distribution has a long list of people who have contributed in many different ways to the Root system.

JAS

JAS is an "Open Source" project. All of the source code is easily available and we use a Java version of make that allows you to build the system yourself, in the same way on any platform, using the simple instructions on our web site. (Building JAS from scratch takes less than one minute, and much less if you only need to recompile files you have changed). Any changes or additions you make are likely to be happily accepted back into the project.

In addition you can often extend JAS without having to learn the internals of the program by writing a plugin which adds the extra functionality you require.

The developers have made most decisions (on direction) with feedback from people using JAS for specific experiments (particularly LCD, Babar). We have a mailing list and bug report page and encourage feedback and suggestions from anyone (negative feedback is very welcome, especially if accompanied by suggestions for improvements).

Questions from the audience:

Q: What licences are needed by these software suites for HEP?

A: (Fons) In ROOT we have an Open Source approach.

A: (Bob Jacobsen, on behalf of Tony Johnson) We use an Open Source/GPL approach.

A: (Andreas) For the packages developed by us, we use an Open Source approach. For the commercial components, you have to simply register with us if you are part of the CERN research programme, otherwise you have to negociate your own licence. The exact type of licence for the packages developed by us is not yet defined, will be either GPL or something very similar.

Q: To what extent is CINT a constraint for people willing to develop new features in ROOT? If I want to write an extension to a ROOT class, should it be CINT complaint?

A: (Fons) There are no constraints as far as CINT is concerned. You only need to have code that compiles on all the platforms supported by ROOT (C++ standard compiler).

HOW DOES THE SOFTWARE UTILIZE LARGE PARALLEL FARMS FOR COMPUTATION?

JAS

JAS has been designed from the outset to run in a "client-server" mode, and to support distributed data analysis. There are Java bindings to many of the GRID components (e.g. GLOBUS) and we expect that features of the GRID such as global authentication will be easy to

interface to JAS. We believe that the model of moving the code to the data (rather than vice-versa) is most applicable to HEP data, and think Java is the best language for exploiting this due to its high performance and built-in network and code portability features. It has not yet been tested with very large datasets on large farms, but that will hopefully be done in the coming year.

ROOT

Following our long experience with parallel architectures in the early 1990s and in particular the development of the PIAF system for PAW, we developed a first prototype of PROOF in 1997 with the goal of using a parallel cluster in a heterogeneous environment. We are investing a lot of effort with PROOF to support large parallel farms. This work will be gradually integrated with the current GRID projects (e.g. Alice has submitted a GRID proposal based on PROOF).

LHC++/Anaphe

This has not yet been studied in the context of Anaphe/LHC++/Lizard. The analysis/design/implementation of this will start at the beginning of 2001 based on the Globus GRID toolkit.

CHOICE OF SCRIPTING LANGUAGE

To what extent can a user or experiment choose their scripting language (e.g. Java, Python, CINT, etc)? Can an experiment choose more than one?

JAS

First, Java is NOT a scripting language. Scripting languages are designed differently from compiled languages such as Java, C++ and Fortran, and to use a compiled language as a scripting language or vice-versa would be unwise. Having said that Java does exhibit some of the advantages sometimes associated with scripting languages, such as very fast compile, load, run cycle (especially when using dynamic loading to load only your analysis routines, as in JAS).

We are currently adding support for scripting languages to JAS; we made a demonstration of beanshell as a scripting language during the school. There are many other scripting languages available for Java, including JPython, a complete and very fast implementation of Python in Java. Any Java scripting language can be very easily used with JAS (or any Java program). There is no technical reason why an experiment should not use more than one.

ROOT

The default command/script interface in Root is based on CINT. If a user does not like CINT, he can make an interface to other languages such as Python. The Root classes may be invoked directly. It is worth mentioning that the number of requests for this solution is close to 0.

Subir Sarkar (L3 Bombay and CDF) has developed an interface to the histogramming package using the Java native interface (JNI) and also a native Java interface to the Root files.

LHC++/Anaphe

In Lizard, the scripting language can be any of those supported by SWIG (www.swig.org) including Perl, Python, Tcl/Tk, Mzscheme, Ruby and Guile. A module for Java (although this is not a scripting language) is in preparation.

QUESTION FROM TONY JOHNSON TO THE STUDENTS:

Suppose that during the CSC 2000, the authors of Root, JAS and LHC++ get together and decide they are wasting time creating similar Analysis systems. Over a late night Ouzo they all decide to work together on their next project, a system for analysing and predicting the future value of the Euro.

They all immediately quit HEP to form their new start-up company and become fabulously wealthy. You are assigned to provide support for one of these orphaned analysis packages for the next 5 years of your experiment (15 years if you are working on an LHC experiment). Which one would you choose?

Votes:

ROOT 25

JAS 17

LHC++ 4 (of which 2 were LHC++ developers)

The other students did not vote. Bob Jacobsen completed this discussion on issues covered by asking a few questions to the audience:

Are we ready for the LHC?

What's not yet understood?

What's still to be built?

SOFTWARE ENGINEERING READING LIST

GENERAL

Software Engineering; Ian Summerville; Addison-Wesley; 1995; ISBN 0-201-42765-6

A Discipline for Software Engineering; Watts S. Humphrey; Addison-Wesley; 1995; ISBN 0-7515-1493-4

Managing the software process; Watts S. Humphrey; Addison-Wesley; 1989; ISBN 0-201-18095-2

Principles of Software Engineering Management; Tom Gilb; Addison-Wesley; 1988; ISBN 0-201-19246-2

R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw Hill, 4th Edition, 1996, ISBN: 0070521824

The mythical man-month; Frederick P. Brooks, Jr.; Addison-Wesley; 1982; ISBN 0-201-00650-2

Software Quality; Joc Sanders & Eugene Curran; Addison-Wesley; 1994; ISBN 0-201-63198-9

OO ANALYSIS AND DESIGN

Principles of Object-Oriented Analysis and Design; James Martin; Prentice Hall; 1993; ISBN 0-13-720871-5

Object-Oriented Modeling and Design; James Rumbaugh et al., Prentice Hall; 1991; ISBN 0-13-629841-9

Object-Oriented Modelling with Syntropy; Steve Cook & John Daniels; 1994; ISBN 0-13-203860-9

Building Object Applications That Work; Scott W. Ambler; 1998; ISBN 0-521-64826-2

UML

Real-Time UML; Bruce Powel Douglass; 1998; ISBN 0-201-32579-9

The Unified Software Development Process; I. Jacobson, G. Booch, J. Rumbaugh; 1998; ISBN 0-201-57169-2

UML Distilled; Martin Fowler; 1997; ISBN 0-201-32563-2

Applying Uml and Patterns; Craig Larman; ISBN 0137488807

H.E. Eriksson & M. Penker, UML Toolkit, Wiley, 1998, ISBN: 0471191612

UML 2001: A Standardization Odyssey, Cris Kobryn, Comms. Of the ACM, Oct. 1999/Vol. 42, No. 10

ONLINE REFERENCES

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

Rational Corporation http://www.rational.com

Software Development Magazine http://www.sdmagazine.com

Software Engineering Resources http://www.rspa.com/spi/

Dr. Dobbs Software Tools for the Professional Programmer http://www.ddj.com

Journal of Object Oriented programming http://www.joopmag.com/

OBJECT-ORIENTED DESIGN AND IMPLEMENTATION

Makoto Asai
Hiroshima Institute of Technology, Hiroshima, Japan
(Geant4 Collaboration)

Abstract
This lecture introduces some basic concepts of Object-Orientation and designing and implementing a program based on Object-Orientation. These concepts are more important than the detailed syntaxes of a language and they will guide you to learn C++/Java as a language which stands on Object-Orientation.

1. INTRODUCTION

If you are writing your code which is exclusively used by yourself and it will be used within a temporary short duration, you can ignore the quality of your code. But you are developing your code with your colleagues and/or your code will be used by your collaborators for years, you should be aware of “good software”. Good software is

- Easy to understand the structure
- Easy to find/localize/fix a bug
- Easy to change one part without affecting to other parts
- Well modularized and reusable
- Easy to maintain and upgrade
- etc.

Object-Orientation is a paradigm which helps you to make a good software.

Use of so-called “Object-Oriented language” such as C++ or Java does not guarantee the Object-Oriented Programming. Badly written C++/Java code is worse than badly written Fortran code. Well- designed, Object-Oriented good software can be relatively easily implemented by using Object-Oriented language. In this sense language is a tool to realize Object-Orientation. This lecture introduces some basic concepts of Object-Oriented Programming. These concepts are more important than the detailed syntaxes of a language and these concepts will guide you to learn C++/Java as a language which stands on Object-Orientation.

Object-Oriented Programming (OOP) is the programming methodology of choice in the 1990s. OOP is the product of 30 years of programming practice and experience.

- Simula67
- Smalltalk, Lisp, Clu, Actor, Eiffel, Objective C
- and C++, Java

OOP is a programming style that captures the behavior of the real world in a way that hides detailed implementation. When successful, OOP allows the problem solver to think in terms of the problem domain. Three fundamental ideas characterize Object-Oriented Programming.

- Class/Object, Encapsulation
- Class hierarchies, Inheritance

- Abstraction, Polymorphism

In this lecture, it must be noted that all sample codes are written in C++.

2. CLASS/OBJECT AND ENCAPSULATION

2.1 Class and object

Object-Oriented Programming (OOP) is a data-centered view of programming in which data and behavior are strongly linked. Data and behavior are conceived of as classes whose instances are objects. OOP also views computation as simulating behavior. What is simulated are objects represented by a computational abstraction.

The term abstract data type (ADT) means a user-defined extension to the native types available in the language. ADT consists of a set of values and a collection of operators and methods that can act on those values. Class objects are variables of an ADT. OOP allows ADT to be easily created and used. For example, integer objects, floating point number objects, complex number objects, four momentum objects, etc., all understand addition and each type has its own way (implementation) of executing addition. An ADT object can be used in exactly same manner as a variable of native type. This feature increases the readability of the code.

```
FourMomentum a, b, c;
c = a + b;
```

In OOP, classes are responsible for their behavior. For example, the *FourMomentum* class should have the following definitions to ensure the above equation could be implemented.

```
class FourMomentum
{
 public:
  FourMomentum(double px, double py, double pz, double e);
  ~FourMomentum();
 public:
  FourMomentum& operator = (const FourMomentum & right);
  FourMomentum operator + (const ThreeMomentum & right);
        ....
```

2.2 Encapsulation

Encapsulation consists of the internal implementation details of a specific type and the externally available operators and functions that can act on objects of that type. The implementation details should be inaccessible to client code that uses the type. It is mandatory to make data members private and provide public Set/Get methods accessible to them. Also, make all Get and other methods which do not modify any data member “cosnt”. The “const” methods can be accessed even for constant ADT objects. Strict use of constant ADT objects allows you the safe programming.

Changes of the internal implementation should not affect on how to use that type externally. Following two implementations of *ForuMomentum* should be used exactly same manner from any other code. Figure 1 is an example class diagram taken from Geant4 toolkit[1]. *G4Track* and *G4Step* are the classes which hide actual data quantities and provide some public methods to the respective private data members.

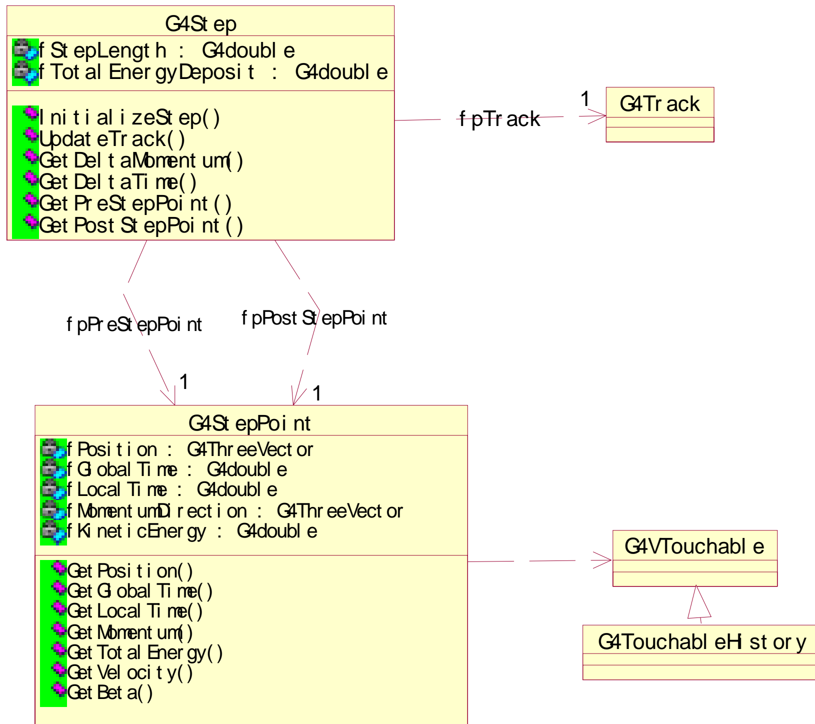


Figure 1. A class diagram taken from Geant4 toolkit which demonstrates encapsulation

```
class FourMomentum                class FourMomentum
{                                 {
  private:                          private:
    double m_Px;                      double m_P;
    double m_Py;                      double m_Theta;
    double m_Pz;                      double m_Phi;
    double m_E;                       double m_E;
  public:                           public:
    void SetP(double p);              void SetP(double p);
    double GetP() const;              double GetP() const;
....                              ....
```

3. CLASS HIERARCHIES AND INHERITANCE

Inheritance is a mean of deriving a new class from existing classes, called base classes. The newly derived class uses existing codes of its base classes. Through inheritance, a hierarchy of related types can be created that share codes and interfaces. A derived class inherits the description of its base class. Inheritance is a method for copying with complexity. Figure 1 is an example of class hierarchy.

It is better to avoid protected data members. Data members in a base class should be private and protected non-virtual access methods to them should be provided. Also, it is advised to avoid

unnecessary deep hierarchies. For example, should a trajectory class and a detector volume class be derived from a single base class, even though both of them have a “Draw()” method? Following the naïve concepts that everyone can easily understand leads the well-designed and thus easy-to-maintain code. Also, it is advised to avoid unnecessary multiple inheritance. In many cases, delegation can solve the problem. Figure 3 is an example of delegation.

Type-unsafe collection is quite dangerous.

- In C++ case, pointer collection of void or very bogus base class.

- In Java case, default vector collection of “Object” base class.

Type-unsafe collection easily reproduces the terrible difficulties we experienced with the Fortran common block.

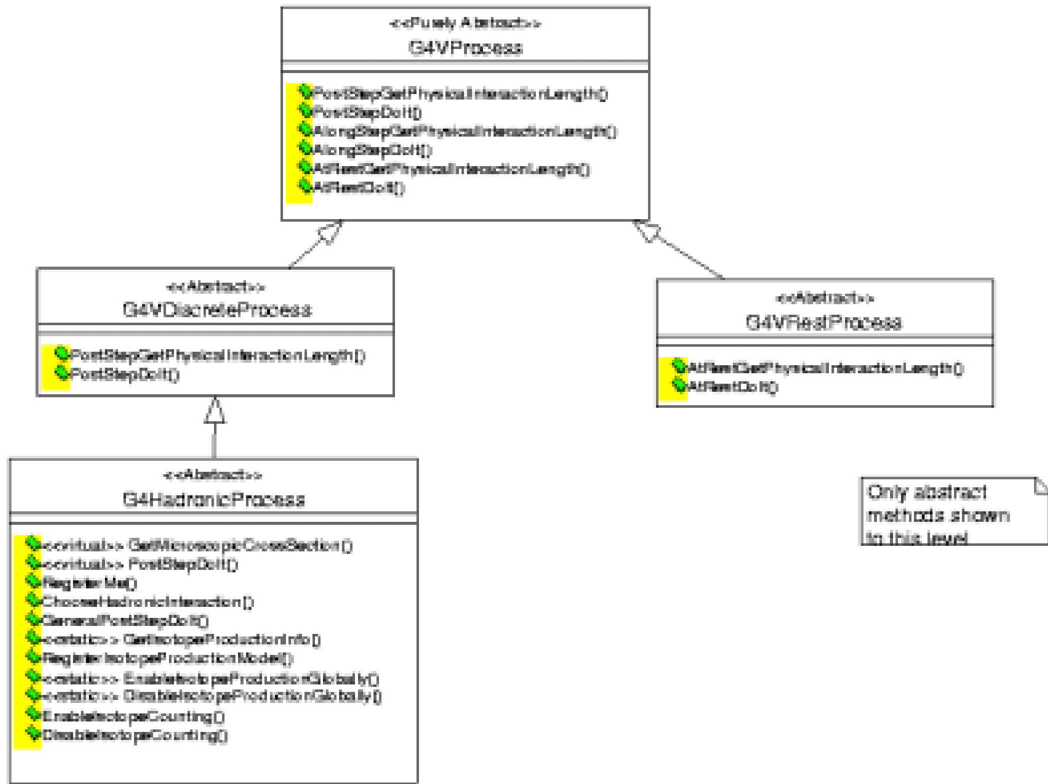


Figure 2. A class diagram taken from Geant4 toolkit which demonstrates class hierachy

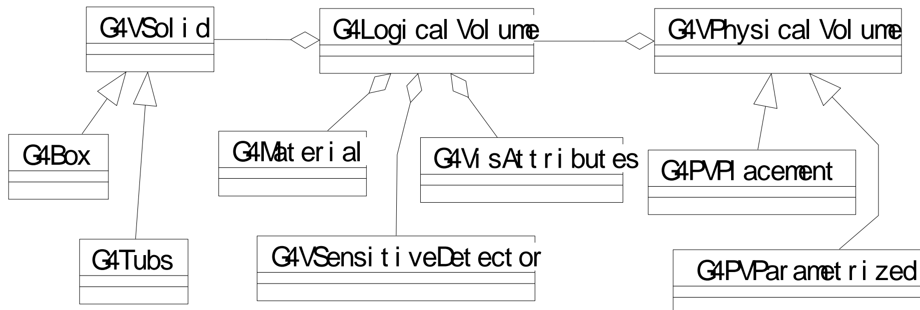


Figure 3. A class diagram taken from Geant4 toolkit which demonstrates delegation

4. ABSTRACTION AND POLYMORPHISM

Abstraction and Polymorphism enable “Rapid Prototyping”. High level class diagrams and scenario diagrams should be made first before going to the detailed design/implementation of actual concrete classes.“ Proof of concepts” demonstration must be done with just a couple of concrete classes (or just one dummy concrete class) for each abstract base class.

Abstraction and polymorphism localizes responsibility for an abstracted behavior. They also help the modularity and portability of the code. For example, Geant4 is free from the choice of histogramming or persistency techniques. Also, GUI and visualization in Geant4 are completely isolated from Geant4 kernel via the abstract interfaces.

Polymorphism has lots of forms.

- Function and operator overloading
- Function overriding
- Parametric polymorphism

In C++, an operator is overloadable . A function or an operator is called according to its signature, which is the list of argument types. If the arguments to the addition operator are integral, then integer addition is used. However, if one or both arguments are floating point, then floating point addition is used. Operator overloading helps the readability.

```
double p, q, r;
r = p + q;
FourMomentum a, b, c;
c = a + b;
```

Using virtual member functions in an inheritance hierarchy allows run-time selection of the appropriate member function. Such functions can have different implementations that are invoked by a run-time determination of the subtype, which is called virtual method invocation or dynamic binding.

```
G4VHit* aHit;
for(int i = 0; i < hitCol->entries(); i++)
{
  aHit = (*hitCol)[i];
  aHit->Draw();
}
```

In this code sample, “Draw()” method is virtual. Thus the way how actual hits will be drawn is not (or should not) known at this code. The way should differ for each type of hit (e.g. calorimeter or drift chamber) and it should be implemented in the individual hit concrete class.

Functions of same name are distinguished by their signatures. For the case of function overloading of “non-pure virtual” virtual functions, all (or none) of them should be overridden. It must be noted that overriding “overrides” overloading. This is an intrinsic source of a bug even though compiler warns. Following sample gives an output of “*Int*” instead of “*Double*”.

```
class Base {
  public:
    Base() {;}
    virtual void Show(int i) { cout << "Int" << endl; }
    virtual void Show(double x) { cout << "Double" << endl; }
}
Class Derived : public Base {
  public:
    Derived() {;}
    virtual void Show(int i) { cout << "Int" << endl; }
}
main() {
  Base* a = new Derived();
  a->Show(1.0);
}
```

C++ also has parametric polymorphism, where type is left unspecified and is later instantiated. STL (Standard Template Library) helps a lot for easy code development.

5. UNIFIED SOFTWARE DEVELOPMENT PROCESS

A software development process is the set of activities needed to transform a user's requirements to a software system (see figure 4). The Unified Software Development Process (USDP)[2] is a software development process which is characterized by Use-case driven, Architecture centered and Iterative and incremental.

There are many different types of requirements.

- Functional requirements
- Data requirements
- Performance requirements
- Capacity requirements
- Accuracy requirements
- Test/Robustness requirements
- Maintainability, extensibility, portability, etc., "ability" requirements

All of these requirements drive use-cases and architectures.

Figure 4. Software process

A software system should be used by the user. Thus the developer of the system must know the users' needs. The term user refers not only to human users but also to other system which interacts with the system being developed. An interaction from/to the user is a use-case. A use-case is a piece of functionality in the system which captures a requirement.

The role of software architecture is similar in nature to the role of architecture plays in building construction. A plan of building is looked at from various viewpoints, such as structure, services, heat conduction, electricity. This allows the builder to see a complete picture before actual construction. The software architecture must be influenced by the requirements of both use-case dependent and use-case independent. Architecture is not a framework.

The Unified Modeling Language (UML)[3] has various diagrams. Not the all diagrams must be used for the software development. But it must be stressed that many of them are quite useful for the Object-Oriented software design and implementation. In this lecture, details of UML diagrams are not covered but the lecture in this school given by R. Jones[4] gives the detail.

A software process has four steps, Object-Oriented Analysis, Object-Oriented Design, Implementation and Test. These steps must be taken many turns (so-called spiral approach) instead of once (so-called water fall approach). The first turn should be the "proof of concepts" demonstration. User's requirements document and diagrams should be regularly updated during the turns.

6. SUMMARY

Basic concepts of Object-Oriented Programming are more important than the detailed syntaxes of a language and these concepts will guide you to learn C++/Java as a language which stands on Object-Orientation. Well-designed, Object-Oriented good software can be relatively easily implemented by using Object-Oriented language.

ACKNOWLEDGEMENTS

The author wishes to thank the organizers of the school for their supports on this lecture.

REFERENCES

[1] Geant4 Home page. http://cern.ch/geant4/

[2] The Unified Software Development Process, I. Jacobson, G. Booch, J. Rumbaugh; 1998, ISBN 0-201-57169-2

[3] The Unified Modeling Language Reference Manual, I. Jacobson, G. Booch, J. Rumbaugh; 1998, ISBN 0-201-30998-X

[4] R. Jones, "Software Design" Lecture of CSC 2000.

OO DESIGN AND IMPLEMENTATION: JAVA AND JAVA ANALYSIS STUDIO

A.S. Johnson
Stanford Linear Accelerator Center, P.O. Box 4349, Stanford, CA 94309, USA

Abstract
We present an overview of the Java language, Java virtual machine, and the large set of standard libraries and extensions available for Java, and provide a discussion of issues that effect Java performance. This is followed by an introduction to Java Analysis Studio (JAS), a tool written in Java, and in which Java is the language used to perform data analysis. We will also explore how OO techniques have been used to built a system from modular components including visualization, fitting and data-access components that can be used together, or on their own.

1. JAVA

1.1 Introduction to Java

James Gosling first developed Java in 1991 at Sun[1]. In was initially designed as a small language targeted at consumer electronics devices (cable boxes, VCR's, smart toasters etc). It had little success in this arena, but in 1994, the HotJava web browser appeared[2], written in Java and with the then unique ability to execute "applets" – small programs written in Java and downloaded into the browser via the web. Amongst much hype Java was adopted as the "Web Programming Language" and licensed from Sun by many key Internet companies including Netscape, Microsoft, Oracle, *etc*.

Over the past five years Java has come a long way from its beginnings as a cool language for embedding spinning coffee cups in web pages. Nowadays it is a mature OO language, with a large set of powerful standard libraries, which allow graphical programs to run without change across most platforms, including Unix, PC, Mac, *etc*. Although Java's role in the web browser has diminished it has become the dominant language in the development of server side web applications, and in terms of demand for programmers now meets or exceeds the demand for C++ programmers. Java has made inroads in many other arenas, including scientific programming[3].

The primary motivation behind the design of Java is reliability; the intention being that the compiler should find as many errors as possible, and once successfully compiled the program should have a high probability of running as intended. In comparison C++ has been designed with efficiency as its number one priority, thus the user is given a lot of control over implementation details: Objects on the stack *vs.* on the heap, manual memory allocation/deallocation *etc*. The downside of this is that the user (especially the naïve user) has many opportunities for subtle errors, often resulting in hard to debug problems. As we shall see later, modern optimization techniques applied to Java code mean that the advantage in performance gained by C++ is often very small.

Where the functionality of C++ and Java overlap the Java syntax is normally identical to the C++ syntax, however many of the historic features of C++, or features judged to add excessive complexity for small gains, have been dropped. For example Java has:

- No pointers – instead object "references" are used. References are similar to pointers except that there is no pointer arithmetic, and no explicit dereference operators.
- No explicit delete operator, instead objects are freed when they are no longer reachable (via a process known as garbage collection).

- No operator overloading.
- No global variables or functions, everything in Java is part of a class or interface.

1.2 Java “Virtual Machines”

The Java language is designed to be used in a slightly different way than languages such as Fortran or C/C++. The Java compiler does not directly convert Java source code into machine specific instructions, instead it generates machine independent “pseudo-machine-code” called Java bytecodes. To run a Java program you must have a Java “Virtual Machine” (VM) on the target machine. The earliest Java VM’s simply interpreted the bytecodes at runtime to execute the program, although more modern Java VM’s convert the bytecodes to machine code at runtime, often employing sophisticated optimization techniques which we will describe in more detail later.

1.3 Java Libraries

In addition to the Java compiler and Java virtual machine, the third component of the Java “platform” is the large set of machine independent standard libraries supplied with Java or available from third parties. The large set of libraries make programming in Java more efficient in two ways, first the developer does not have to waste time reimplementing common functionality, and secondly it is much easier to reuse components written by someone else since they are unlikely to be based on some incompatible graphics or utility library.

Some examples of common Java libraries are the Collections library, the Swing GUI library, 2D and 3D graphics libraries, JDBC and ODMG database interface libraries, and libraries for distributed programming including CORBA and RMI.

1.4 Java Performance

Optimization of C++ and Fortran programs is typically done at compile time. Static optimization as this process is now called is a very mature technology developed over many decades. Over the past five years a lot of research has been done on dynamic optimization[4,5], a technique better suited to languages such as Java, resulting in very impressive performance improvements over earlier versions of Java.

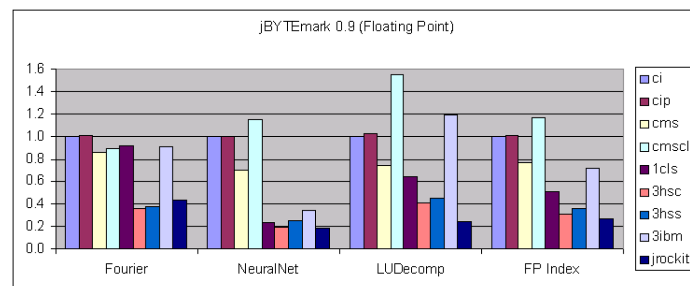


Fig. 1 Results of a benchmark comparing Java and C++ performance.
For a detailed explanation see ref. [6].

Dynamic optimization is performed by the virtual machine at run-time. The program is continuously monitored as it executes, and only code that is actually found to be a bottleneck is optimized. Since dynamic optimization is performed at run-time, more information is available to the optimizer than in the case of compile time optimization. For example, a dynamic compiler is able to generate machine code optimized for the actual processor being used (e.g. Pentium, Pentium II, Pentium III etc), and only generate thread safe code if the program is running in a multi-threaded environment. In addition, the optimizer knows whether a particular method has been overridden by any loaded subclasses, and can eliminate virtual call overhead in many cases. Since Java is a dynamic

language, new classes can be loaded at any time, so the optimizer has to be smart enough to invalidate and recompile code segments that were compiled under conditions that are no longer hold.

Because of dynamic optimization the performance of Java programs have steadily improved and have now become very close to that of C++ programs. Recent benchmarks[6] show Java performance averaging about 60% of C++ performance (see Fig. 1), although with wide fluctuation about this average, probably due to the relative immaturity of the dynamic optimizers technique. Further improvements in Java performance can be expected as the optimizers mature.

2. JAVA ANALYSIS STUDIO

2.1 What is Java Analysis Studio?

Java Analysis Studio is a desktop data analysis application aimed primarily at offline analysis of high-energy physics data. The goal is to make the application independent of any particular data format, so that it can be used to analyze data from any experiment. The application features a rich graphical user interface (GUI) aimed at making the program easy to learn and use, but which at the same time allows the user to perform arbitrarily complex data analysis tasks by writing analysis modules in Java. The application can be used either as a standalone application, or as a client for a remote Java Data Server. The client-server mechanism is targeted particularly at allowing remote users to access large data samples stored on a central data center in a natural and efficient way.

2.2 Graphical User Interface

When the application is first started, it presents the user with the interface as shown in Fig. 2. The goal is to present the user with a consistent interface from which all analysis tasks can be performed. The graphical user interface also features a complete help system, wizards to help new users get started, facilities for viewing and manipulating plots, and is extensible via Plugins written in Java to provide user or experiment specific features.

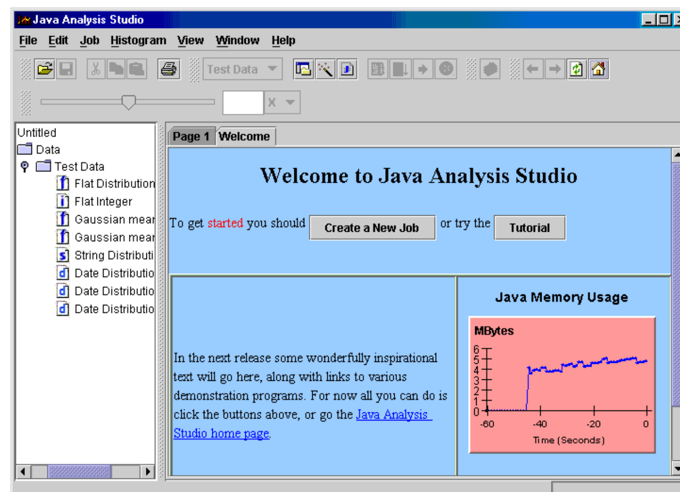


Fig. 2 The Java Analysis Studio Graphical User Interface

2.3 User Analysis Modules

Although some other graphical analysis environments allow users to define their analysis by wiring together pre-built analysis modules, we believe that the complexity of real-life physics analysis problems quickly makes such approaches unworkable. Although JAS allows some simple analysis operations to be performed using the graphical user interface, serious analysis is done by writing analysis modules in Java. To this end JAS includes a built-in editor and compiler (Fig. 3) so that analysis modules can be written, compiled, loaded and run from within the JAS environment. One of

the advantages of Java is that it supports dynamic loading and unloading of classes, which results in a very short compile-run-debug cycle that is excellent for data analysis tasks.

Java Analysis Studio is provided with a package of classes for creating, filling and manipulating histograms that can be used from the user's analysis module. Binning of histograms is delegated to a set of partition classes, which allows great flexibility in defining different types of built-in or user-defined histograms. The built in partition classes support histograming dates and strings as well as integers and floating-point numbers, and also support either traditional HBOOK style binning while filling, or delayed binning which allows histograms to be rebinned and otherwise manipulated using the GUI after they have been filled.

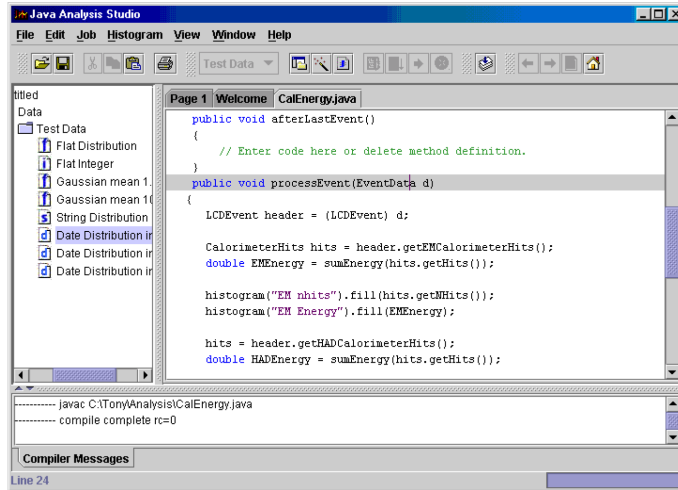


Fig. 3 JAS includes a built-in compiler and editor.

2.4 Data Formats

Unlike most other data analysis applications which force the user to first translate the data into a particular format understood by that application, Java Analysis Studio is able to analyze data stored in almost any format. It does this by requiring that for any particular data format an interface module be available which can provide the glue between the application and the data. The application is distributed with several built-in Data Interface Modules (DIMs), which provide support for paw n-tuples, hippo n-tuples, SQL databases (implemented using Java's JDBC database interface), StdHEP files and flat-file n-tuples.

Support is provided for analyzing either n-tuples or arbitrarily complex object hierarchies. While analyzing n-tuple data a number of graphical user interface options are available for plotting columns of data singly or in pairs, as well as applying cuts. The intention is to provide an interface similar to that provided by HippoDraw[7]. While analyzing n-tuple data can sometimes be convenient it is also rather limiting, and therefore we also support analysis events consisting of arbitrarily complex trees of objects.

JAS can read data stored on the user's local machine, or stored on a remote data server. The application has been designed from the outset with this client-server approach in mind, and as a result the interface that is presented to the user is identical whether the data being analyzed is stored locally or on a remote server. When running in client-server mode the user's analysis modules are still edited and compiled locally, but when run the analysis modules are sent over the network and executed on the data server.

Since the analysis modules are written in Java and compiled into machine independent class files it is easy to move them from the users machine to the remote data server. The Java runtime

provides excellent built-in security features to prevent user analysis modules from interfering with the operation of the data server on which they are running. When the user requests to see a plot created by an analysis module, only the resulting (binned) data is sent back over the network, resulting in a very modest bandwidth and latency requirements even when analyzing huge data sets. Due to its modest network requirements JAS works quite well even when accessing a remote data server via a 28.8 kb modem.

It is hoped that the client-server features built into Java Analysis Studio will prove particularly useful to researchers who typically access data from universities where it is not possible to store the Petabyte sized data samples typically generated by today's HEP experiments. Using Java Analysis Studio such researchers can still take advantage of the powerful graphical features of their desktop machines, while analyzing data which is stored remotely. The performance of JAS is such that it is quite possible to forget that the data is not located on the local machine.

2.5 Histogram and Scatterplot Display

One of the key components of Java Analysis Studio is the JASHist bean, which is responsible for the display of histograms and scatter plots. The charts are very efficient at redrawing themselves, so that they can easily display rapidly changing data. By interacting with the GUI, end users can easily change the title or legends just by clicking on them and typing new information, and can change the range over which data is displayed just by clicking and dragging on the axes (Fig. 4).

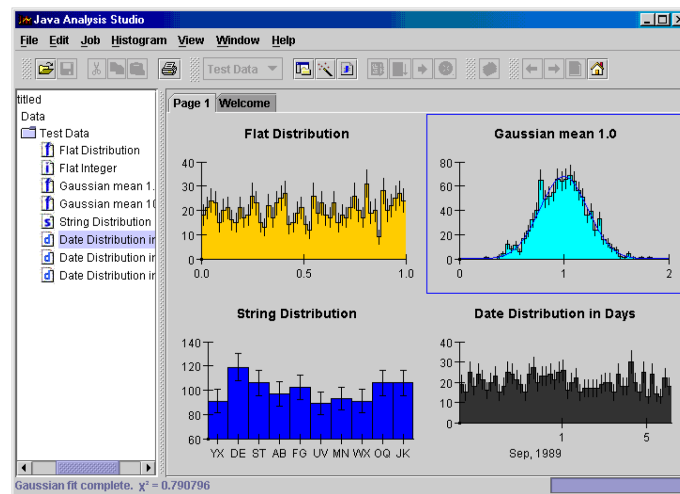


Fig. 4 Histogram Display in JAS

The JASHist bean is designed using the model-view-controller pattern, so that data to be displayed need only implement a simple Java interface and need have no other dependence on the JAS package. This makes interfacing arbitrary data to the plot bean very straightforward. Care has been taken in the design and implementation of the JASHist bean to ensure that it is a modular component that can be used easily in other applications.

The current JASHist bean includes support for:

- Display of 1-D histograms, 2-D histograms and scatter plots. Scatter plot support is optimized to handle up to millions of points.
- Overlaying of several histograms or scatter plots on one plot.
- Interactive fitting of arbitrary functions to 1-D histograms.
- Numeric or time axes, plus axes with named bins.
- Many display styles that can be set interactively or programmatically.

- Dynamic creation and display of slices and projections of 2-D data.
- Direct user interaction, by clicking and dragging.
- Data that is constantly changing, including very efficient redrawing to support rapidly changing data (handles over 100 updates/second).
- Printing using both Java 1 and Java 2 printing models. High quality print output is available when using Java 2.
- Saving plots as GIF images or as XML. Support for encapsulated postscript and PDF is in progress.
- Custom overlays that allow data to be displayed using user defined plot routines for specialized plots.

2.6 Extending Java Analysis Studio

Java Analysis Studio is designed to be extended by end users and/or by experiments. A number of API's have been defined to make it possible to build extensions without having to understand the details of the Java Analysis Studio implementation. Currently supported extension API’s include:

- The Data Interface Module API for writing interfaces to new data types.
- The Function API for writing new functions.

The Fitter API for writing new fitters. This allows user defined fitters to be used in place of the built-in least squares fitter. (A fitter which used Java's Native Interface to call Minuit would be a useful extension).

A plugin API for writing user or experiment specific extensions to the JAS client. The Plugin API allows extensions to be tightly integrated into the client GUI by by supporting creation of new menu items, creation of windows and dialogs, and interaction with the event stream. The above shows two plugins that have been developed for Linear Collider Detector studies, one shows the MC particle hierarchy as a Java tree, and the other is a simple event display. A recent plugin allows a full WIRED event display to run within JAS (Fig. 5).

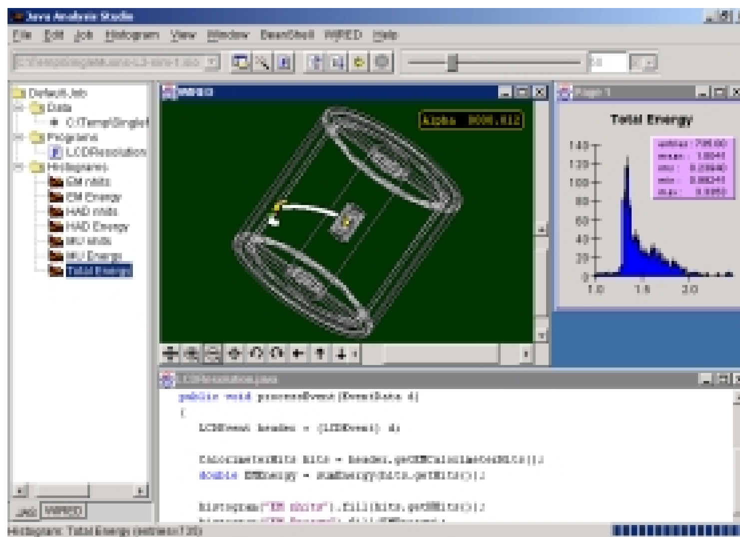


Fig. 5 The WIRED event display running as a JAS plugin.

2.7 Implementation

The application has been built as far as possible on industry standards and using commercial components where consistent with the goal of making the final application redistributable with no runtime license fees.

2.8 Open Source Model

JAS is now an open source project, with source code browsable directly from the JAS web site (using jCVS servlet), or accessible using any CVS client. The instructions for gaining read-only access to the CVS repository are available on the JAS web site, and read-write access is available to registered developers. Our intention is to continue to refine the design of JAS to make it easier to integrate with other applications and our hope is that making the source available will make it easier for other to understand how it works, and to contribute fixes and improvements.

In order to further facilitate cross-platform development we have adopted jmk, a pure Java utility similar to make. This enables JAS to be built on any platform with a Java Development Kit (JDK) available.

2.9 Availability

At the time of writing Java Analysis Studio version 2.2.1 is available for download from our web site at: http://jas.freehep.org. Currently we *support* for Windows (NT/95/98/2000), Linux and Solaris, however since the application is written entirely in Java (except for the optional Paw, Hippo and Stdhep DIM's) it should work on any platform with a JDK 1.1 (or greater) compliant Java Virtual Machine.

REFERENCES

[1] "A Brief history of the Green Project" - http://java.sun.com/people/jag/green/index.html.
[2] "Java Technology – An Early History" - http://java.sun.com/features/1998/05/birthday.html.
[3] Information on scientific programming in Java can be found at http://www.javagrande.org. For Java use in High Energy Physics see http://java.freehep.org.
[4] "IBM System Journal on Java Performance" http://www.research.ibm.com/journal/sj39-1.html.
[5] "Java Hotspot Dynamic Optimizer" http://java.sun.com/products/hotspot/whitepaper.html.
[6] "The Java Performance Report" http://www.javalobby.org/features/jpr/.
[7] "HippoDraw as Electronic Notebook", http://www-sldnt.slac.stanford.edu/hepvis/Papers/Web/5/.

INTRODUCTION TO GEANT4

Makoto Asai
Hiroshima Institute of Technology, Hiroshima, Japan
(Geant4 Collaboration)

Abstract

Geant4 is one of the first successful attempts to re-design a major package of CERN software for the next generation of HEP experiments using an Object-Oriented environment. It is being developed and maintained by more than 100 colleagues of all over the world with adoption of the most recent software engineering methodologies, choice of Object-Orientation and C++. In this paper, emphasis is paid on how these modern techniques especially Object-Orientation affect on the design, implementation and maintenance of Geant4.

1. INTRODUCTION

1.1 Geant4 collaboration

Geant4 [1] is the successor of GEANT3, which is well-known as the world-standard toolkit for HEP detector simulation. Geant4 is one of the first successful attempts to re-design a major package of CERN software for the next generation of HEP experiments using an Object-Oriented environment. A wide variety of requirements also come from heavy ion physics, CP violation physics, cosmic ray physics, medical applications and space science applications. In order to meet such requirements, a large degree of functionality and flexibility are provided. G4 is not only for HEP but goes well beyond that.

At the beginning of 1990's, GEANT3 faced on two major difficulties on its maintenance and further development.

i) Because of too complex structure driven by too many historical reasons, it became impossible to add a new feature or to hunt a bug. This indicated the limitation of FORTRAN and/or old style of programming.

ii) Shortage of manpower at CERN became serious. This indicated the limitation of "central center" supports.

To solve these difficulties, a new R&D project (CERN RD44 [2]) was started in December 1994. This R&D project is made of a world-wide collaboration and decided the adoption of the most recent software engineering methodologies and choice of Object-orientation and C++.

This R&D project was successfully terminated in December 1998 with the first production release of Geant4. Now, the Geant4 collaboration becomes MoU-based collaboration, which is maintaining its code, documents and examples, at the same moment of upgrading its functionalities. The collaboration will continue to maintain and upgrade Geant4 for decades, e.g. lifetime of LHC.

1.2 Performance? Flexibility? Usability?

We believe that Geant4 is a fundamental test of the suitability of the object-oriented approach and C++ language for software in HEP, where performance is an important issue. As a consequence, Geant4 releases should be regularly monitored against the performance provided by Geant3 at comparable physics accuracy. For the case of geometrical navigation, Geant4 automatically optimizes the user's geometrical description, while the user had to implement his/her geometry very carefully to

exploit the full performance of Geant3 navigation. And Geant4 provides faster navigation than optimized Geant3 descriptions. For the case of electro-magnetic physics in a simple sampling calorimeter, we confirmed Geant4 is 3 times faster when using the same cuts (in the sensitive material) as GEANT3. And more than a factor 10 faster when seeking the best performance in Geant4 that maintains constant the quality of the physics results as Geant3. Geant4 is faster than GEANT3 in all aspects, when its power and features are well exploited.

The flexibility of Geant4 could be emphasized in many aspects. For example in physics processes, much wider coverage of physics comes from mixture of theory-driven, cross-section tables, and empirical formulae. Thanks to polymorphism mechanism, both cross-sections and models can be combined in arbitrary manners into one particular process. We have wide variety of processes for various kinds of particles such as slow neutron, ultra-high energy muon, optical photon, parton string models, shower parameterization, event biasing technique, etc., and new areas are still coming. Also thanks to the abstraction and polymorphism, we have many types of geometrical descriptions such as CSG, BREP and Boolean, and all the geometrical descriptions are STEP compliant. Because events and tracks are treated as class objects, overlapping events, suspension of slow looping tracks and postponing these tracks to next event, and priority control of tracks in the stacks could be realized without losing any performance overhead. We must stress that everything in Geant4 is open to the user so that he/she can choose physics processes and models, integrator in magnetic/electric field, GUI and visualization technologies, and histogramming and persistency mechanism according to his/her environment or preference.

In terms of usability, User Requirements Document (URD) states many different use-cases from various fields. Thanks to the inheritance mechanism, the user can derive his/her own classes easily. Many abstract layers and default behaviors are provided at the same time. Many reusable examples and documents are provided and are still continuously evolving with the user's contribution. Geant4 had already started to be used by various scientific fields well beyond HEP.

2. OBJECT-ORIENTED ANALYSIS AND DESIGN ADOPTED IN GEANT4

2.1 User Requirements Document

As mentioned in the previous chapter, Geant4 has to satisfy a wide variety of requirements. At the beginning of the development of Geant4, requirements and use-cases were gathered from contributors who belonged to wide variety of scientific fields. These requirements and use-cases were studied one by one, categorized and collected into Geant4 User Requirements Document (URD) [3]. All the top level designs of Geant4 based on URD.

URD is the base of Geant4. It has been being maintained through the life of Geant4. Once a new requirement comes and it is recognized as reasonable, URD is updated to get it in at the same moment of its implementation. Global design and its implementation are regularly checked for their consistencies with respect to URD.

2.2 Category Design

Figure 1 shows the top level categories of Geant4. Dashed allows show the dependencies between categories. With respect to the URD, 17 major categories were found by Object-oriented analysis and design. The most important issue of designing top level categories is establishing well-defined category boundaries with respect to the mandates of each category, and defining clear "use relations" between categories with avoiding loop dependency. For example, "Processes" category uses "Particle" via "Track", but "Material" category should not know about "Particle".

2.3 Basic concepts in Geant4

Geant4 is the Object-Oriented toolkit which provides functionalities required for simulations in HEP and other fields. Simulation is a "virtual reality". Simulation is used both to help designing detectors

during R&D phase and understanding the response of the detector for the physics studies. To create such virtual reality we need to model the abstract behaviors of the particle-matter interactions, geometry and materials in order to propagate elementary particles into the detector. We need also to describe the sensitivity of the detector for generating raw data. In this section, some basic concepts used in Geant4 are explained. These concepts are quite important for understanding the structure and the behavior of Geant4.

2.3.1 Run

As an analogy of the real experiment, a run of Geant4 starts with "Beam On". Within a run, the user cannot change neither detector geometry nor settings of physics processes. This means detector is inaccessible during a run. Conceptually, a run is a collection of events which share the same detector conditions.

2.3.2 Event

At beginning of processing, an event contains primary particles. These primaries are pushed into a stack. When the stack becomes empty, processing of an event is over. G4Event class represents an event. It has following objects at the end of its processing.

- List of primary vertexes and particles
- Trajectory collection (optional)
- Hits collections
- Digits collections (optional)

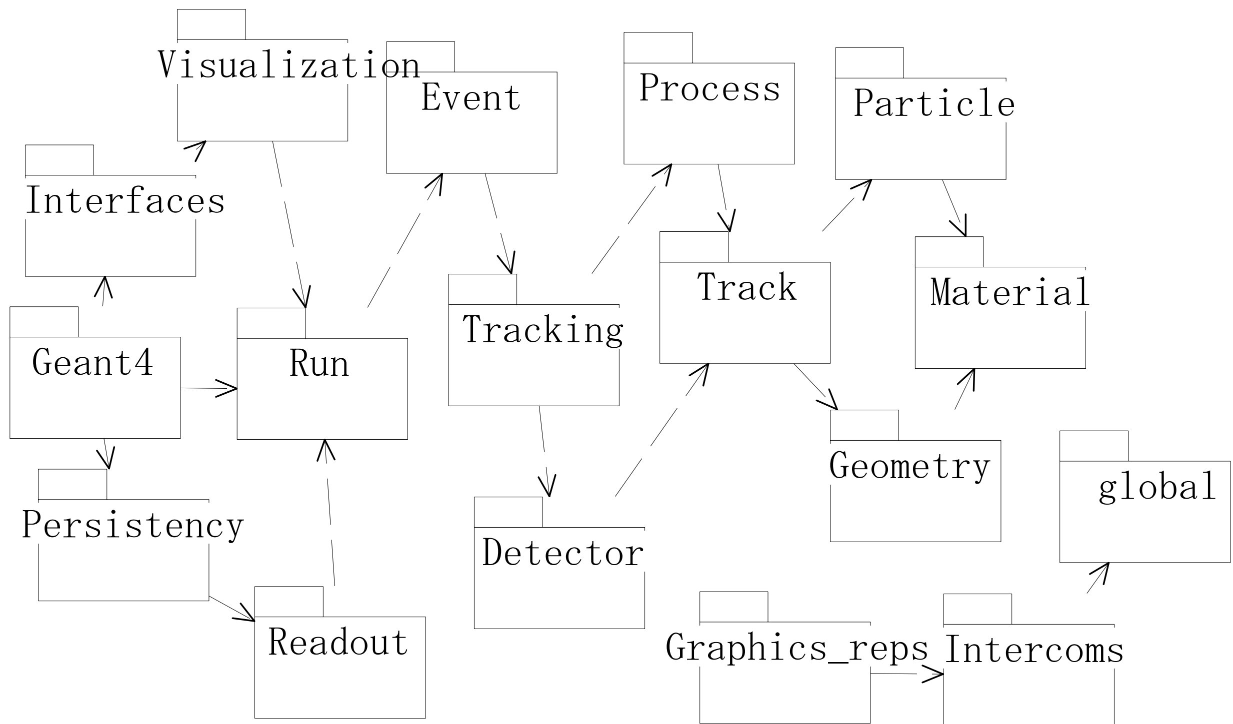


Fig.1 Category diagram of Geant4.

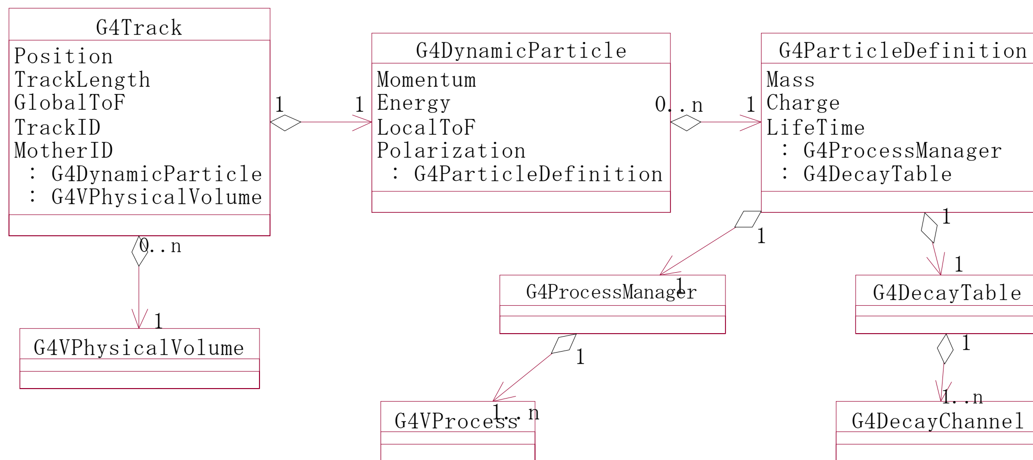


Figure 2. Classes which represent a track

2.3.3 Track

Track is a snapshot of a particle. Step is "delta" information to a track. Thus, track is not a collection of steps. Track is deleted when

- it goes out of the world volume,
- it disappears (e.g. decay),
- it goes down to zero kinetic energy and no "at rest" additional process is required,
- the user decides to kill it.

As shown in Figure 2, a track in Geant4 is represented by three layers of class objects. First two objects of G4Track and G4DynamicParticle classes are unique for each track but an object of G4ParticleDefinition is shared by all tracks of same type.

2.3.4 Step

Step has two points and also "delta" information of a particle (energy loss on the step, time-of-flight spent by the step, etc.). Each point knows the volume. In case a step is limited by a volume boundary, the end point physically stands on the boundary, and it logically belongs to the next volume.

2.3.5 Trajectory

Trajectory is a record of a track history. It stores some information of all steps done by the track as objects of G4TrajectoryPoint class. It is advised not to store trajectories for secondary particles generated in a shower because of the memory consumption. The user can create his own trajectory class deriving from G4VTrajectory and G4VTrajectoryPoint base classes for storing any additional information useful to the simulation.

2.3.6 Physics processes

Geant4 has three basic types of physics processes as follows.

- "At rest process" (e.g. decay at rest) which is applied only for a particle at rest.
- "Continuous process" (e.g. ionization) which is accumulatively applied along a step of a particle.

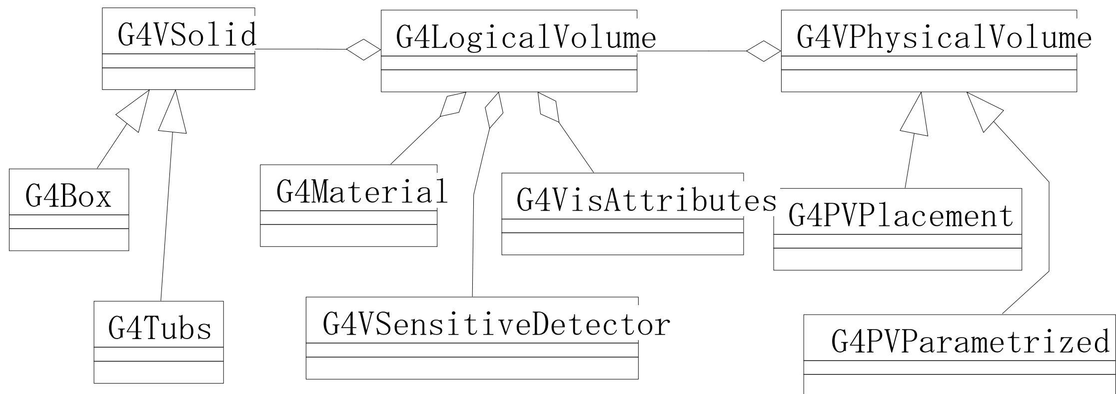


Figure 3. Classes which represent geometry

- "Discrete process" (e.g. decay on the fly) which is applied at the end point of a step.

Transportation is still a process which interacts with volume boundary. A process which requires the shortest physical interaction length limits the step.

In Geant4, the user defines cut-off by length instead of energy. It makes poor sense to use the energy cut-off. Range of 10 keV gamma in Silicon is around a few cm, while range of 10 keV electron in Silicon is a few micron. In Geant4, cut-off represents the range of secondaries. It does not mean that the track is killed at the corresponding energy. A track is traced down to zero kinetic energy except it meets to the other condition listed in 2.3.3. Additional "AtRest" process may occur even if the track becomes at rest. In case the energy corresponding to the given cut-off in a thin material is less than the available energy range of a physics process, Geant4 will not stop that particle by that process in the current volume (material). In case the track goes into another volume (material) which is more dense, that process may stop the track.

2.3.7 Geometry

Figure 3 illustrates three conceptual layers of classes for geometrical description.

- G4VSolid has shape and size. Various shapes and types of solid classes are derived from G4Vsolid base class.
- G4LogicalVolume has daughter volumes, material, sensitivity, user limits, etc.
- G4VPhysicalVolume has position and rotation. Objects of G4VPhysicalVolume can share an object of G4VPhysicalVolume if only their positions/rotations are different from each other.

2.3.8 Sensitive detector and hit

Each "Logical Volume" can have a pointer to a sensitive detector. Hit is a snapshot of the physical interaction of a track or an accumulation of interactions of tracks in the sensitive region of your detector. A sensitive detector creates hit(s) using the information given in G4Step object. The user has to provide his/her own implementation of the detector response. Hit objects, which still are the user's class objects, are collected in a G4Event object at the end of an event.

2.4 Basic scenario how Geant4 runs

Geant4 has two major phases in its execution. The first phase is initialization, which is triggered by *Initialize()* method of G4RunManager, or an equivalence of UI command. Figure 4 is a scenario diagram of initialization phase. In this phase, construction of material and geometry according to the user's concrete implementation of G4VUserDetectorConstruction, and construction of particles,

physics processes and calculation of cross-section tables according to the user's concrete implementation of G4VUserPhysicsList occur.

The second phase is run, which is triggered by *BeamOn()* method of G4RunManager, or an equivalence of UI command. Figure 5 is a scenario diagram of the run phase. The run phase starts with closing and optimizing the geometry and them the event loop follows. Figure 6 shows the scenario flow inside the event loop.

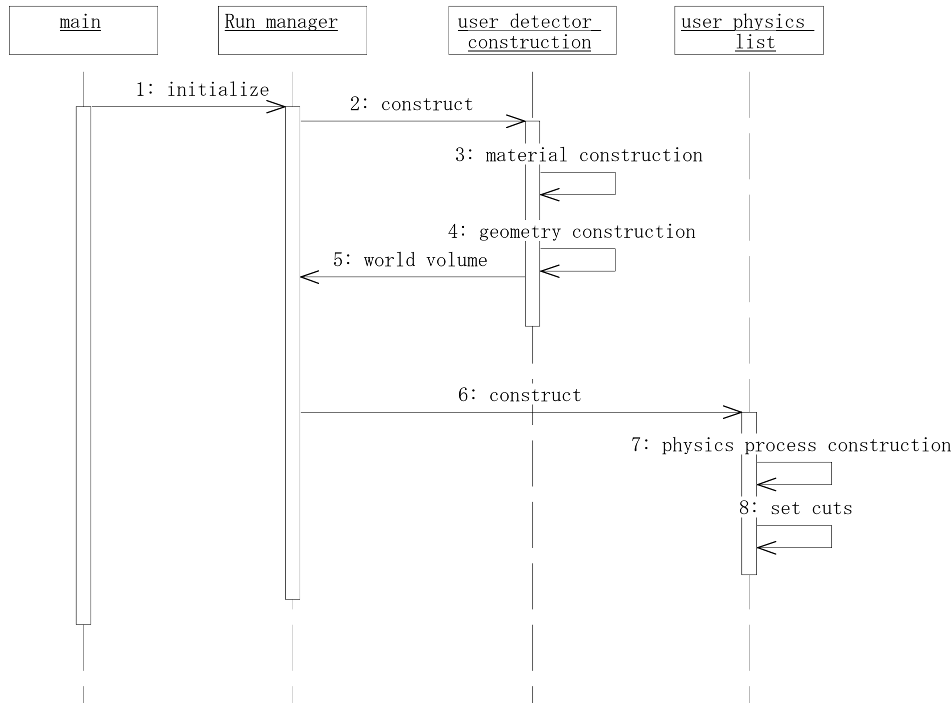


Figure 4. Scenario diagram for the initialization phase

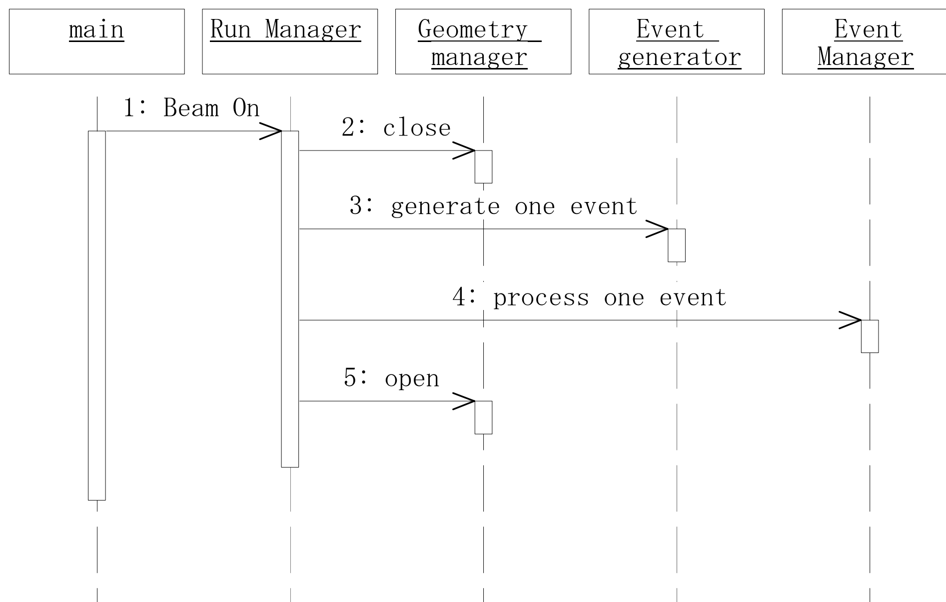


Figure 5. Scenario diagram for an event loop

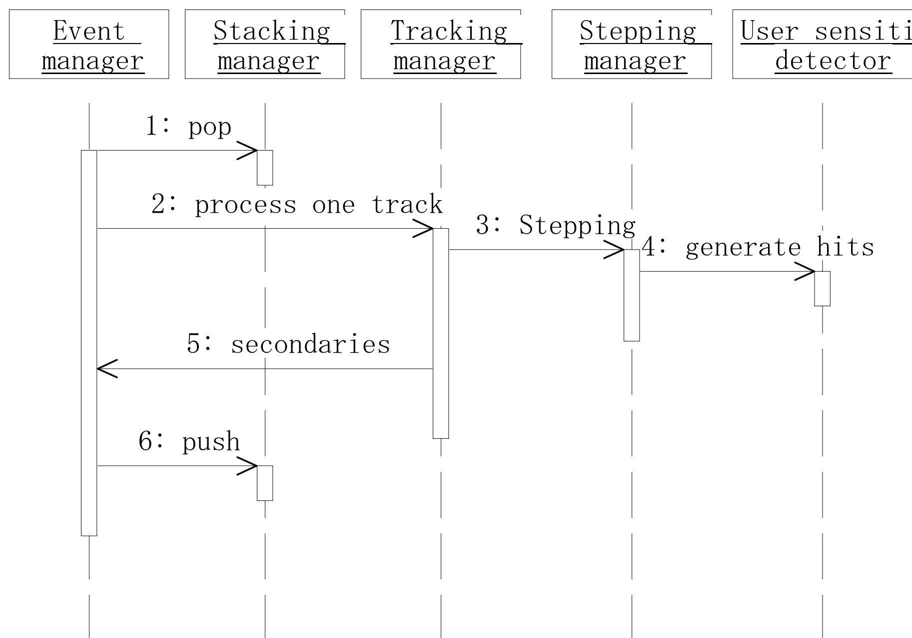


Figure 6. Scenario diagram for processing of one event

2.5 Unit system

Geant4 has no default unit. To give a number, unit must be "multiplied" to the number. For example :

double width = 12.5*m;

double density = 2.7*g/cm3;

Almost all commonly used units are available. The user can define new units. To get a number in a particular unit, divide a variable by the unit.

G4cout << dE / MeV << " (MeV)" << G4endl;

3. THE MINIMAL THINGS TO USE GEANT4

In this chapter, one mandatory method and three mandatory classes that the user has to provide are explained. These mandatory classes must be derived from the abstract base classes provided by Geant4 and they must be set to G4RunManager.

3.1 *main()*

Geant4 does not provide the *main()* method. The user has to write his/her own *main()* method and, in it, the user has to construct G4RunManager or its derived class, and has to set three mandatory classes inherit G4VUserDetectorConstruction, G4VUserPhysicsList and G4VUserPrimaryGeneratorAction. The user can define Visualization Manager, (G)UI session, optional user action classes, and/or persistency manager in his/her *main()* to customize the behavior of application based on Geant4.

3.2 Detector construction

The user has to derive his/her own concrete class from G4VUserDetectorConstruction abstract base class. In the virtual method Construct(), the user has to

- Construct all necessary materials

- Construct volumes of his/her detector geometry

- Construct his/her sensitive detector classes and set them to the detector volumes

Optionally the user can define visualization attributes and/or step limits of his/her detector elements.

3.3 Physics list

Geant4 does not have any default particles or processes. Even for the particle transportation, the user has to define it explicitly. To define particles and processes, the user has to derive his/her own concrete class from G4VUserPhysicsList abstract base class. In the concrete implementation, definition of all necessary particles and all necessary processes must be declared and proper processes must be assigned to proper particles. Geant4 provides lots of utility classes/methods to make the user be easy to define. Also in this concrete implementation, cut-off values must be declared.

3.4 Definition of primary generators

The user has to derive his/her concrete class from G4VUserPrimaryGeneratorAction abstract base class and to pass a G4Event object to one or more primary generator concrete class objects which generate primary vertices and primary particles. Geant4 provides two generators, those are G4ParticleGun and G4HEPEvtInterface, which interfaces to /hepevt/ common block via ascii file. An interface directly to C++ version of PYTHIA direct interface will be available quite soon. Also, an interface to HepMC is planned.

4. OPTIONAL USER ACTION CLASSES

Geant4 provides five user's optional "action classes" to control the behavior of Geant4. In this chapter, these classes are explained with probable usages. All user action classes, methods of which are invoked during "Beam On", must be constructed in the user's *main*() and must be set to the RunManager.

4.1 G4UserRunAction

This base class defines two methods, *BeginOfRunAction(const G4Run*)* and *EndOfRunAction(const G4Run*)*. These are invoked at the beginning and the end of each run. Most probable use-case of these methods are booking and manipulating histograms.

4.2 G4UserEventAction

This base class defines two methods. The first method is *BeginOfEventAction(const G4Event*)* and it is invoked at the beginning of each event processing but after the generation of the primary particles. Thus the G4Event object already has primary particles and they can be used for the event filtering. The second method is *EndOfEventAction(const G4Event*)*, which is invoked at the end of each event. Basic analysis of the event can be implemented in this method.

4.3 G4UserStackingAction

This base class defines three methods. The user can customize the stacking mechanism by implementing his/her own concrete class derived from this base class. In Geant4, three are three stacks, which are called as "Urgent stack", "Waiting stack" and "Postpone to next event stack", respectively. When a new track is pushed to the G4StackManager, regardless of it is primary or secondary, the method *ClassifyNewTrack(const G4Track*)* is invoked. The user can classify the track as one of the following four classifications.

Urgent : the track to be sent to the "Urgent stack"

Waiting : the track to be sent to the "Waiting stack"

PostponeToNextEvent : the track to be sent to "Postpone to next event stack"

Kill : the track to be killed immediately without pushing to a stack

The track is always popped from "Urgent stack" and sent to G4EventManager. Once this stack becomes empty, the method *NewStage()* is invoked. The user can re-classify the tracks stacked in the "Waiting stack".

4.4 G4UserTrackingAction

This base class defines two methods, *PreUserTrackingAction(const G4Track*)* and *PostUserTracking Action(const G4Track*)*, which are invoked at the beginning and the end of each track processing, respectively.

4.5 G4UserSteppingAction

This base class defines one method of *UserSteppingAction(const G4Step*)* which is invoked at the end of procedure of each step. The user can kill the track, suspend the stepping and send it back temporary to the stack, or suspend the stepping and postpone it to the next event.

5. USER SUPPORTS

5.1 Documents

Geant4 provides various documents. All of them are accessible from the Geant4 official Web page[1]. The documents and examples are continuously evolving. Currently, the site has six documents as follows.

- Introduction to Geant4 : This document
- Installation guide : This document explains how to install Geant4 toolkit onto various types of machines, at the same time required external libraries and the way of getting them are summarized.
- User's guide for application developer : This documents explains the basic usage of Geant4 toolkit suitable for the user who is developing a simulation program based on Geant4.
- User's guide for toolkit developer : This documents explains how to add/alternate some built-in functionalities of Geant4, such as adding a new physics process, adding a new shape of solid, etc.
- Physics reference manual : This document explains in deep what kinds of physics processes are implemented in Geant4 toolkit and what are the referenced materials.
- Software reference manual : This document lists all of Geant4 class definitions. This document is available only on Web and required page is generated automatically by access.

5.2 Examples

Geant4 provides various examples. All of them are kept updated every release and continuously evolving. Especially, examples for new areas of applications rather than high-energy physics are glowing according to the number of the users and collaborators in these areas. There are three categories of examples.

- Novice examples : These examples are aimed to learn how to use Geant4 toolkit.
- Extended examples : These examples demonstrate how to use Geant4 toolkit with some other external packages such as histogramming, persistency, etc.
- Advanced examples : Each example in this category demonstrates how to use one or two category in Geant4 in detail. Some examples in this category are converted from our global system tests.

5.3 Other user supports

In our Web site, the bug reporting system is provided. Once a user encounters to a bug, he/she can report it to this system. The coordinator of corresponding category will investigate the report and the

reporter will get feedback within a few days. Web-based source code browser and the user's news group will be in public quite soon.

6. SUMMARY

Geant4 is one of the first successful attempt to re-design a major package of HEP software for the next generation of experiments using an Object-Oriented environment. A variety of requirements also came from heavy ion physics, CP violation physics, cosmic ray physics, medical applications and space science applications. Geant4 is the first software development in HEP which fully applies most recent software engineering methodologies. Because of adoption of Object-orientation, wide coverage of physics processes, geometrical implementations, etc. is realized. At the same moment, user's extension to area is easily achieved. Geant4 provides various user supports and they are continuously evolving. Geant4 is not only for HEP but goes well beyond that. Intensive use of Geant4 in various fields has already been started.

ACKNOWLEDGEMENTS

The author wishes to thank the organizers of the school for their supports on this lecture. The author wishes to thank Dr. Gabriele Cosmo (CERN/IT) and Dr. Marc Verderi (LPNHE) their most useful comments on this lecture and also their assists on the associated exercises. The author acknowledges all of collaborators of Geant4 for their individual excellent works. All the members of the collaboration who should share the authorship of this lecture are listed on our Web site.

REFERENCES

[1] Geant4 Home page. http://cern.ch/geant4/

[2] S. Giani, et. al. "Geant4: An Object-Oriented toolkit for Simulation in HEP", CERN/LHCC 98-44.

[3] K. Amako, et. al. Users requirements document. The latest version can be found from the Geant4 Web site.

AN INTRODUCTION TO THE GLOBUS TOOLKIT

Giovanni Aloisio
University of Lecce, Lecce, Italy

Massimo Cafaro
University of Lecce, Lecce, Italy

Abstract
The Globus toolkit is a grid middleware being developed at the Institute of Sciences Information of the University of Southern California and at Argonne National Laboratories. In this paper we briefly introduce the core services and the functionalities provided; in particular we address the Globus Security Infrastructure, the resource management and the Grid Information Service.

1. INTRODUCTION

In the past, high performance applications ran on a single supercomputer; advances in both networking technologies and metacomputing environments are going to change this, enabling new classes of applications that can benefit from metacomputing.

The physical infrastructure is now widely available in the United States and is becoming widespread in Europe, although some efforts were undergone to improve the networks available. Using such high speed networks it is feasible to link together supercomputers, data archives, advanced visualization devices such as a Cave, and scientific instruments. But to really exploit these new interesting possibilities the heterogeneous and dynamic nature of the metacomputing environment must be taken into account.

Globus is a hierarchical, layered set of high level services built using a low level toolkit. Its purpose is to provide the middleware needed for grid and metacomputing applications, providing the foundations for building computational grids [1] linking together thousands of resources geographically distributed.

2. THE GLOBUS TOOLKIT

The term metacomputing [2] refers to computation on a virtual supercomputer assembled connecting together different resources like parallel supercomputers, data archives, storage systems, advanced visualization devices and scientific instruments using high speed networks that link together these geographically distributed resources.

We may want to do so because it enables new classes of applications [3][4] previously impossible and because it is a cost effective approach to high performance computing; we can classify the new applications as follows:

- desktop supercomputing;
- smart instruments;
- collaborative environments;
- distributed supercomputing.

Desktop supercomputing includes applications that couple high end graphics capabilities with remote supercomputers and/or databases; smart instruments are scientific instruments like microscopes, telescopes and satellites that require supercomputing power to process the data produced in near real time.

In the class of collaborative environments we find applications in which users at different locations can interact together working on a supercomputer simulation; distributed supercomputing finally is the class of application that require multiple supercomputers to solve problems otherwise too large or whose execution is divided on different components that can benefit from execution on different architectures.

The challenges to be faced before grid computing and metacomputing can be really exploited however include the following issues:

- scaling and selection;
- heterogeneity;
- unpredictable structure;
- dynamic behavior;
- multiple administrative domains.

Scaling is a concern, because we expect that metacomputing environments in the future will become larger, and resources will be selected and acquired on the basis of criteria like connectivity, cost, security and reliability.

Such resources will show different levels of heterogeneity, ranging from physical devices to system software and schedulers policies; moreover traditional high performance applications are developed for a single supercomputer whose features are known a priori, e.g. the latency; in contrast metacomputing applications will run in a wide range of environments thus making impossible to predict the structure of the computation.

Another concern is related to the dynamic behavior of the computation [5], since we cannot be assured that all of the system characteristics stay the same during the course of computation, e.g. the network bandwidth and latency can widely changes, and there is the possibility of both network and resources failure.

Finally, since the computation will usually span resources at multiple administrative domains, there is not a single authority in charge of the system, so that different scheduling policies and authorization mechanisms must be taken into account.

Globus is designed to tackle this issues, and its usefulness in the context of metacomputing was demonstrated in the I-WAY experiment [6][7] and in the Gusto testbed; among the distributed simulations that have been run using Globus there is SF-Express [8], the largest computer simulation of a military battle involving more that 100,000 entities.

3. GLOBUS SERVICES

Globus provides different services in order to enable the metacomputing environment:

- resource management;

- communication;
- information;
- security;
- health and status;
- remote data access;
- executables management.

Resource management is the aim of the GRAM (Globus Resource Allocation Manager) module [9][10]; it takes care of resource location and allocation; moreover it performs process management. The user can specify its application requirements using RSL, the resource specification language.

Resource location is needed, since in a dynamic environment applications cannot know in advance the exact location of the required resources. A broker is in charge of locating and acquiring (allocation) the resources by scheduling them and performing the required initialization.

The NEXUS library of communications [11][12][13] provide unicast and multicast communication services; moreover, different quality of service parameters [14] are taken into account, e.g. jitter, latency, bandwidth. The library can be exploited for the implementation of message passing [15], rpc, distributed shared memory, etc; it uses mechanism for negotiating the best communication method among parties and provide asynchronous remote service requests (RSR).

Using a RSR, a handler on a remote machine can be invoked supplying a typed buffer of data in input. The advantage over traditional rpc mechanism is that the remote applications can be multithreaded thus avoiding blocking.

The Metacomputing Directory Service (MDS) [16] provides a uniform resource information service that allow distributed access to structure and data information about the metasystem status. The module allow both posting and receiving information, using the LDAP protocol (Lightweight Directory Access Protocol).

The Globus Security Infrastructure (GSI) module [17][18][19] is responsible for handling the authentication mechanism used to validate the identity of the users and resources; it makes use of both the Generic Security Service api (GSS) and of SSL (Secure Socket layer). Certificates are used as the principal authorization mechanism.

Fault detection and monitoring of health and status of the metasystem components is possible by using the tools provided by the Heart beat Monitor (HBM) module [20]. Using the HBM a process can be monitored and periodic heartbeats can be sent to one or more monitors.

Remote access to data via sequential and parallel interfaces can be obtained exploiting the Global Access to Secondary Storage (GASS) module [21]; supported operations include remote read, write and append. The Remote I/O (RIO) library [22] implements the MPI I/O interface.

The Globus Executable Management (GEM) is responsible for performing the necessary initialization and setup for the same executable at different sites; it is used to migrate the executables and eventually to compile the source code on the target machine if this step is needed.

4. THE GLOBUS SECURITY INFRASTRUCTURE

Grid aware applications must be designed bearing in mind since the beginning one of the most important issues, security. Therefore the Globus team developed an authentication system known as gssapi, the Generic Security Service api using a public domain implementation of SSL. This system

utilizes the RSA algorithm, thus employing both public and private keys. Authentication relies on X.509v3 certificates provided by each user in their directories, that identify them to grid resources.

5. SECURE SYSTEMS

Secure systems make it hard for people to do things they are not supposed to do; they are designed so that the cost of breaking any component of the system outweighs the rewards. The cost of breaking systems is measured in money, time and risk, both legal and personal. The benefits of breaking systems are control, money or information that can be sold for money. Therefore, the security of a system should be proportional to the resources it protects.

The term secure systems can be misleading; as a matter of fact it does not implies that systems are either secure or insecure, but it is used to denote systems that were designed with security requirements. It is worth noting here that every system can be broken, given enough time and money.

A Computational Grid consists of a set of valuable resources like parallel supercomputers, workstations, databases and smart instruments, so in order to prevent unauthorized accesses to the Grid we need a strong authentication mechanism. To show how Globus handle authentication, we begin reviewing some of the notions belonging to the theory of Cryptography.

6. CRYPTOGRAPHY

Cryptography [23][24] is the science of *secret writing*; it is a branch of mathematics called *cryptology*. Strictly related to cryptography is *cryptanalysis*, the science of breaking (analyzing) cryptography. Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather one set of techniques.

Confidentiality is a service used to keep the content of information from all but those authorized to have it. *Secrecy* is a term synonymous with confidentiality and privacy. There are numerous approaches to providing confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.

Data integrity is a service which addresses the unauthorized alteration of data. To assure data integrity, one must have the ability to detect data manipulation by unauthorized parties. Data manipulation includes such things as insertion, deletion, and substitution.

Authentication is a service related to identification. This function applies to both entities and information itself. Two parties entering into a communication should identify each other. Information delivered over a channel should be authenticated as to origin, date of origin, data content, time sent, etc. For these reasons this aspect of cryptography is usually subdivided into two major classes: *entity authentication* and *data origin authentication*. Data origin authentication implicitly provides data integrity (if a message is modified, the source has changed).

Non-repudiation is a service which prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary. For example, one entity may authorize the purchase of property by another entity and later deny such authorization was granted. A procedure involving a trusted third party is needed to resolve the dispute.

A common way to protect information is for the sender to *encrypt*, or scramble, information before sending it. The receiver *decrypts*, or unscrambles, the information after receiving it. While in transit, the encrypted information is unintelligible to an intruder. *Encryption* applies a mathematical

function to transform the information, called *plaintext*, in order to make it unintelligible to anyone but the intended recipient. *Decryption* is the opposite process of transforming encrypted information, called *ciphertext*, to recover the original information.

The mathematical function used to encrypt or decrypt information is called a *cryptographic algorithm* or *cipher***.** It is worth noting that, since ciphers algorithms are widely known, confidentiality is based on a number called a *key* to be used with the algorithm to encrypt or decrypt previously encrypted information.

A *symmetric cipher* (Fig. 1) uses the same key at the sending and receiving end; these ciphers are also called *private key* or *secret key* ciphers.

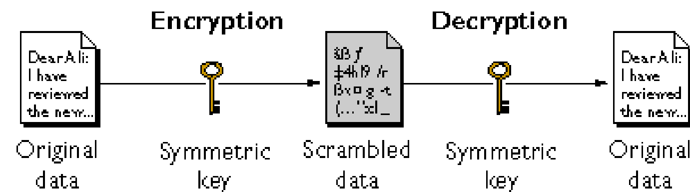


Fig. 1 Symmetric cipher

Symmetric algorithms can be divided into *stream ciphers* and *block ciphers*. Stream ciphers can encrypt a single bit of plaintext at a time, whereas block ciphers take a number of bits (typically 64 bits in modern ciphers), and encrypt them as a single unit. Implementations of symmetric-key encryption can be made highly efficient: the time delay to encrypt or decrypt information is not significant. Moreover, the use of symmetric-key encryption provides a degree of authentication, since information encrypted with one symmetric key cannot be decrypted with any other symmetric key.

Nevertheless, symmetric-key encryption can be effective only if the symmetric key is kept secret by the parties involved. Supposing that anyone else discovers the key, this in turn affects not only confidentiality but also authentication, since people with an unauthorized symmetric key is able to decrypt messages sent with that key and encrypt new messages and send them. This messages appear as if they originate from one of the parties who were originally using the key.

Asymmetric ciphers (Fig. 2), also called *public key* ciphers, make use of a pair of keys - a *public key* and a *private key* - associated with each entity that needs to authenticate itself or to sign or encrypt data. Each public key can be safely published, while the corresponding private key must be kept secret. The mathematical functions associated to public key ciphers are such that data encrypted with a public key can be decrypted only with the corresponding private key.

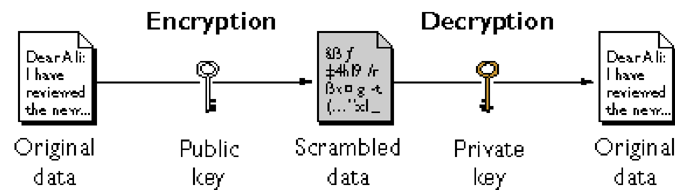


Fig. 2 Asymmetric cipher

To send encrypted data to someone, the information is encrypted using that person's public key. The person who receives the encrypted data decrypts it using the corresponding private key. With asymmetric ciphers there is no need to worry about other people keeping a secret, as in case of symmetric ciphers; asymmetric ciphers can also be used to generate a *digital signature:* data encrypted with your private key can be decrypted only with your public key. Anyway, asymmetric ciphers are computationally expensive.

Before showing the RSA algorithm, we need the following

DEFINITION For $n > 1$, let $\phi(n)$ denote the number of integers in the interval $[1,n]$ which are relatively prime to n. The function ϕ is called the Euler phi function.

FACT (properties of Euler phi function)

(1) If p is a prime, $\phi(p) = p - 1$

(2) If $gcd(m,n) = 1$, $\phi(mn) = \phi(m)\,\phi(n)$

ALGORITHM: Key generation for RSA public key encryption

Each entity creates an RSA public key and a corresponding private key as follows:

1) Generate two large random and distinct primes p and q, each roughly the same size.
2) Compute $n = pq$ and $\phi(n) = (p - 1)\,(q - 1)$
3) Select a random integer e, $1 < e < \phi$, such that $gcd(e, \phi) = 1$
4) Use the extended Euclidean algorithm to compute the unique integer d, $1 < d < \phi$, such that $ed = 1 \pmod{\phi}$
5) The public key is (n,e), the private key is d.

DEFINITION The integers e and d in the RSA key generation are called the *encryption exponent* and the *decryption exponent* respectively, while n is called the *modulus*.

ALGORITHM: RSA public key encryption

Bob encrypts a message m for Alice, which Alice decrypts.

1) Bob obtains Alice's authentic public key (n,e).
2) Represent the message m as an integer in the interval $[0, n - 1]$
3) Compute $c = m^e \bmod n$

4) Send the ciphertext c to Alice

Alice recover plaintext m from c

1) Use the private key d to recover $m = c^d \bmod n$

The strength of encryption is closely related to the difficulty of discovering the key; this in turn depends on both the cipher used and the length of the key. We can think of encryption strength in terms of the size of the keys: in general, longer keys allow stronger encryption of data. Key length is measured in number of bits.

We can now describe the authentication mechanism exploited in the Globus Toolkit, i.e., digital certificates.

7. DIGITAL CERTIFICATES

A *certificate* is an electronic document whose aim is to identify an entity and to associate that identity with a public key. Certificates are similar to a driver's license or a passport, common IDs that provide a generally recognized proof of a person's identity. People who want to get a driver's license apply to a government agency, in this case the Department of Motor Vehicles, which in turn verifies their identity, ability to drive, address, and other information before issuing the license.

People get a certificate in much the same way, applying to a *Certificate authority* (*CA*), an entity that validate identities and issue certificates. CAs can be independent third parties or organizations running their own certificate-issuing server software.

When a CA issues a certificate, a particular public key is *binded* to the name of the entity the certificate identifies. We can say that, for all practical purposes, a certificate is just a public key signed by the issuing CA. This means that the public key certified will work with the related private key owned by the entity identified by the certificate. Since we trust the CA, we trust the identity certified by the CA.

A certificate *always* includes other information like the name of the entity it identifies, the date in which the certificate is issued, an expiration date, the name of the issuer of the certificate, a serial number, etc. Moreover, a certificate must always include the digital signature of the issuing CA.

Certificates are used both for *client authentication,* i.e. the confident identification of a client by a server, and for *server authentication,* i.e. the confident identification of a server by a client. The process of authenticating a user to a server works as follows. The client digitally signs a randomly generated piece of data using the user's private key and sends both the certificate and the signed data to the server. The server authenticates the user's identity using the user's public key found in the certificate.

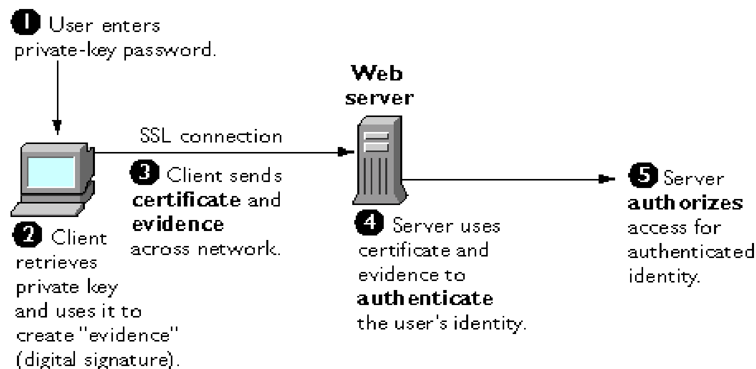


Fig. 3 Certificate based client authentication

Certificate-based authentication is to be preferred to password-based authentication because it is based on what the user has (the private key) as well as what the user knows (the password that protects the private key). The client asks for the password that protects the user's private key the first time the client needs to access it during a given session. After entering this password once, the user doesn't need to enter it again for the rest of the session, even when accessing other SSL-enabled servers.

In the SSL protocol the client uses the private key to sign some data that has been randomly generated on the basis of input from both the client and the server. This data and the digital signature can be used to prove the private key's validity. As a matter of fact, the digital signature can be created only with that private key and can be validated using the corresponding public key against the signed data, which is unique to the SSL session.

7.1 Globus certificates

There exist a number of different certificates to identify several entities. CA certificates are used to identify Cas; both client and server software exploit CA certificates to determine what other certificates can be trusted. Client SSL certificates are used to identify clients to servers using the SSL protocol and server SSL certificates are used to identify servers to clients. Server authentication in SSL may be used with or without client authentication, but server authentication is a requirement for an encrypted SSL session. Client certificates in Globus are compliant to the X.509 v3 certificate specification; an X.509 v3 certificate binds a *distinguished name* (DN) to a public key. A DN is composed of name-value pairs, such as uid = doe, that uniquely identify the certificate *subject*.

The following is an example of DN:

Subject: C=US, O=Globus, O=University of Lecce, OU=HPC Lab, CN=Massimo Cafaro

uid=cafaro,e=massimo.cafaro@unile.it,

cn=Massimo Cafaro,o=University of Lecce, c=IT

The abbreviations shown have these meanings:

- uid: user ID
- e: email address
- cn: the user's common name
- o: organization
- ou: organization unit
- c: country

DNs may also include other name-value pairs.

Every X.509 certificate consists of two sections:

- The data section
- The signature section

7.1.1 The data section

The data section must contains:

- A number that references the version of the X.509 standard supported by the certificate
- The certificate's serial number. Every certificate issued by a CA is required to have a serial number that must be unique among the certificates issued by that CA
- Information concerning the user's public key, including the cipher and a representation of the key itself
- The DN of the CA that issued the certificate
- A starting date and an expiration date that determine the time frame during which the certificate is considered valid
- The DN of the certificate subject, also called the *subject name*
- Optional *certificate extensions*

7.1.2 The signature section

The signature section must contains:

- The cipher used by the CA to digitally sign the certificate
- The CA's digital signature

Here it is an example Globus certificate:

```
issuer :/C=US/O=Globus/CN=Globus Certification Authority
subject:/C=US/O=Globus/O=University of Lecce/OU=HPC Lab/CN=Massimo Cafaro
serial :052B
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 1323 (0x52b)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=US, O=Globus, CN=Globus Certification Authority
    Validity
      Not Before: Sep 27 13:59:59 1999 GMT
      Not After : Sep 26 13:59:59 2000 GMT
    Subject: C=US, O=Globus, O=University of Lecce, OU=HPC Lab, CN=Massimo Cafaro
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public Key: (1024 bit)
```

Modulus (1024 bit):

00:c0:e3:40:15:f2:2e:e7:43:9e:62:e2:5d:aa:ce:

3d:52:9c:dd:7d:6b:d7:59:c0:fe:14:26:3b:83:51:

19:d7:1a:ed:50:ed:e0:9c:52:dd:bb:4b:88:c0:91:

2a:4e:b2:21:e4:be:a3:35:69:59:84:73:c9:8c:cc:

f4:8b:45:4c:79:ef:b5:93:01:3d:ba:15:8d:af:3b:

84:62:56:b8:4f:dd:b9:42:10:93:85:bd:01:e0:85:

d0:09:9a:fb:0c:eb:a0:fe:0d:6f:3e:5f:42:9d:3d:

80:52:18:af:a0:64:2d:b1:76:a4:e9:97:ea:64:ef:

fc:5e:3e:d1:18:11:7d:3e:91

Exponent: 65537 (0x10001)

X509v3 extensions:

Netscape Cert Type:

0x40

Signature Algorithm: md5WithRSAEncryption

4b:22:4e:23:87:cc:30:0b:cf:78:c6:62:99:34:e0:f8:19:7f:

2a:a0:2e:3d:86:82:60:8c:6e:b1:e7:7d:8c:1c:b0:d2:9a:69:

41:27:2b:69:95:9b:7f:07:d9:06:b2:7b:1d:bd:3e:f7:24:35:

c8:60:4b:40:34:03:e5:97:34:0e:5c:82:73:23:f6:ab:43:c2:

34:cd:95:14:99:02:e8:73:63:e2:0f:3a:a1:77:51:f2:de:74:

9b:ff:1d:84:68:46:23:94:a3:c4:ac:c4:a6:7b:ac:bb:13:23:

09:0e:a7:01:b0:17:8b:d1:01:10:fb:6a:2f:32:da:8a:e4:fe:

da:65

-----BEGIN CERTIFICATE-----

MIICOjCCAaOgAwIBAgICBSswDQYJKoZIhvcNAQEEBQAwRzELMAkGA1UEBhMCVV
MxDzANBgNVBAoTBkdsb2J1czEnMCUGA1UEAxMeR2xvYnVzIENlcnRpZmljYXRpb24gQXV0
aG9yaXR5MB4XDTk5MDkyNzEzNTk1OVoXDTAwMDkyNjEzNTk1OVowZzELMAkGA1UEBh
MCVVMxDzANBgNVBAoTBkdsb2J1czEcMBoGA1UEChMTVW5pdmVyc2l0eSBvZiBMZWNjZ
TEQMA4GA1UECxMHSFBDIExhYjEXMBUGA1UEAxMOTWFzc2ltbyBDYWZhcm8wgZ8wDQ
YJKoZIhvcNAQEBBQADgY0AMIGJAoGBAMDjQBXyLudDnmLiXarOPVKc3X1r11nA/hQmO4
NRGdca7VDt4JxS3btLiMCRKk6yIeS+ozVpWYRzyYzM9ItFTHnvtZMBPboVja87hGJWuE/duUIQ
k4W9AeCF0Ama+wzroP4Nbz5fQp09gFIYr6BkLbF2pOmX6mTv/F4+0RgRfT6RAgMBAAGjFTAT
MBEGCWCGSAGG+EIBAQQEAwIAQDANBgkqhkiG9w0BAQQFAAOBgQBLIk4jh8wwC894xm
KZNOD4GX8qoC49hoJgjG6x532MHLDSmmlBJytplZt/B9kGsnsdvT73JDXIYEtANAPllzQOXIJzI/
arQ8I0zZUUmQLoc2PiDzqhd1Hy3nSb/x2EaEYjlKPErMSme6y7EyMJDqcBsBeL0QEQ+2ovMtqK5
P7aZQ==

-----END CERTIFICATE-----

8. RESOURCE MANAGEMENT

Globus provides a flexible Resource Specification Language (RSL) that allows users to express their application's constraints. The resource management architecture (Fig. 4) allow easy allocation and co-allocation of computational resources. The Globus Resource Allocation Manager (GRAM) (Fig. 5) provides such services despite the heterogeneity of the resources managed.

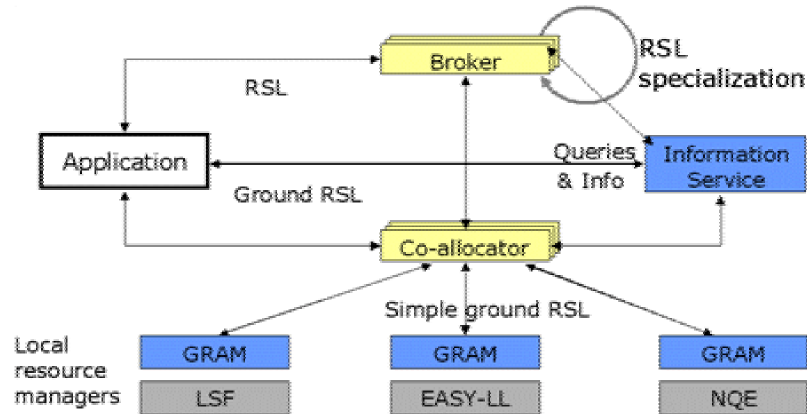


Fig. 4 Resource Management Architecture

RSL syntax closely resembles LDAP filters syntax; using RSL users can easily inquire about resource features like machine type, memory, cpu, clock speed and number of processors available in case of parallel machines. RSL also allow to specify for a job the directory to be used, where the executable can be found, the command line arguments and environment variables that may be needed.

The GRAM allows to run jobs remotely, and provides an api for submitting, monitoring, and terminating jobs. When the user submits a job, a request is generated and forwarded to a daemon called the gatekeeper running on the remote computer. The gatekeeper manages the request creating a job manager responsible for the job. The job manager is in charge of starting and monitoring the remote application, it also communicates job state changes to the user and terminates when its job has terminated execution either normally or by failing.

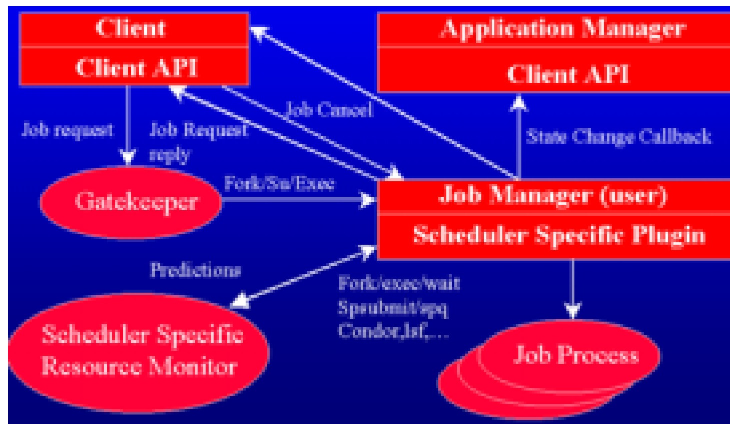


Fig. 5 GRAM

The GRAM supports a multiple queues scheduling model, in which users or resource brokers submit job requests that are registered as pending jobs. A job during its lifetime undergoes state changes according to the state diagram in Fig. 6.

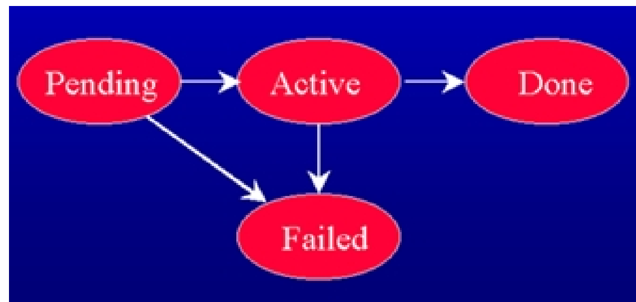


Fig. 6 State transitions of a Globus Job

Globus provides the user with the possibility of submitting a multirequest, i.e., a specification of a job whose execution requires the simultaneous allocation of multiple resources. Co-allocation of resources is managed by the DUROC (Dynamically Updated Request Online Co-allocator) as shown in Fig. 7 and is an atomic operation with a simple semantics. Either all of the requested resources are available and Globus co-allocate them, or none of them get scheduled. Pending request can be edited dynamically by adding new nodes or removing failed ones, and are delayed to last possible minute, using a barrier synchronization mechanism.

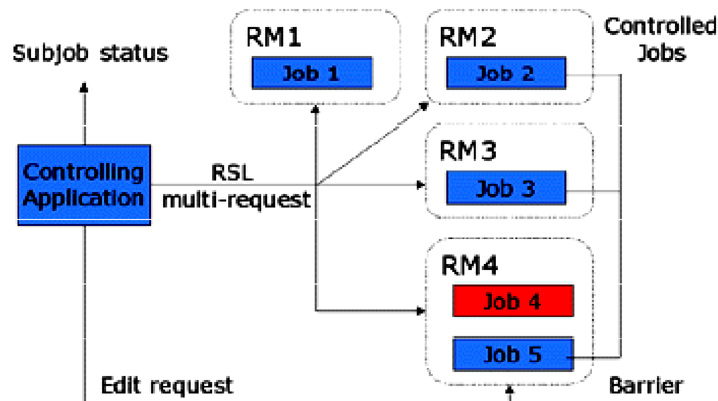


Fig. 7 DUROC Architecture

9. THE GRID INFORMATION SERVICE

The Globus toolkit provides the users with a directory service called Grid Information Service (formerly known as Metacomputing Directory Service or MDS). Aim of this service is to allow grid applications to easily locate and determine features of resources. The GIS can be queried to find where resources with a required architecture, installed software, available capacity, network bandwidth etc. are, or given a specific resource, to find its distinguishing features.

Basically, the users can query a centralized server (this is true for Globus v1.1.2 as of this writing, but the upcoming Globus release v1.1.3 is going to change this, implementing the GIS as a true distributed service, much like the DNS) using the LDAP (Lightweight Directory Access Protocol) api or the globus command *grid-info-search*.

The GIS can be populated with a variety of useful information. This information is published both as a *white pages directory*, in which information is related to a particular *distinguished name*, and as a *yellow pages directory* in which information is categorized using a number of *object classes* defined by the Globus team. These classes are currently outdated, and a new set of schemas is being developed jointly with the US Grid Forum.

The GIS provides a flexible information infrastructure critical to grid applications; such applications should harness the information published in the GIS to determine the available resources and the state of the computational grid. The information could also be exploited to optimize the applications, by tuning them at runtime if a reconfiguration procedure is needed on the basis of both static and dynamic contents.

Information accessible via the GIS include, but is not limited to:

- computing resources features like IP address, installed software, system administrator, networks connected to, OS version, current load;
- network features like bandwidth and latency, protocols and logical topology;
- Globus infrastructure features like hosts and resource managers.

The uniform information infrastructure provided by the GIS is scalable and permits efficient access to dynamic data, moreover it is based on a standard (LDAP).

10. CONCLUSIONS

We have described the core services composing the Globus toolkit, which is becoming the de facto middleware standard to build grid enabled applications. In particular, we addressed the Globus Security Infrastructure explaining related concepts along the way, the resource management architecture and the Grid Information Service.

A complete description of the Globus toolkit is beyond the scope of this paper, nevertheless interested readers will find more information at http://www.globus.org.

REFERENCES

[1] Ian Foster, Carl Kesselman, “The Grid. Blueprint for a new computing infrastructure”, Morgan Kaufmann, San Francisco 1999

[2] C. Catlett, L. Smarr, “Metacomputing”, Communications of the ACM, 35 (1992), pp. 44-52

[3] W. Benger, I. Foster, J. Novotny, E. Seidel, J. Shalf, W. Smith, P. Walker, “Numerical Relativity in a Distributed Environment”, Ninth SIAM Conference on Parallel Processing for Scientific Computing, Apr. 1999

[4] Gregor von Laszewski, Mei-Hui Su, Joseph A. Insley, Ian Foster, John Bresnahan, Carl Kesselman, Marcus Thiebaux, Mark L. Rivers, Steve Wang, Brian Tieman, Ian McNulty, “Real-Time Analysis, Visualization, and Steering of Microtomography Experiments at Photon Sources”, Ninth SIAM Conference on Parallel Processing for Scientific Computing, Apr. 1999

[5] W. Smith, I. Foster, V. Taylor, “Predicting Application Run Times Using Historical Information”, Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.

[6] I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke, “Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment”, Proc. 5th IEEE Symposium on High Performance Distributed Computing. pg. 562-571, 1997

[7] T. DeFanti, I. Foster, M. Papka, R. Stevens, T.Kuhfuss, “Overview of the I-WAY: Wide Area Visual Supercomputing”, International Journal of Supercomputer Applications, 10(2), pp.123-130, 1996

[8] S. Brunett, D. Davis, T. Gottshalk, P. Messina, C. Kesselman, “Implementing Distributed Synthetic Forces Simulations in Metacomputing Environments”, Proceedings of the Heterogeneous Computing Workshop, Mar. 1998

[9] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy, “A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation”, Intl Workshop on Quality of Service, 1999

[10] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, N. Karonis, S. Martin, W. Smith, S. Tuecke, “A Resource Management Architecture for Metacomputing Systems”, Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,1998

[11] I. Foster, J. Geisler, C. Kesselman, S. Tuecke, “Managing Multiple Communication Methods in High-Performance Networked Computing Systems”, Journal of Parallel and Distributed Computing, 40, pp.35-48, 1997

[12] I. Foster, C. Kesselman, S. Tuecke, “The Nexus Approach to Integrating Multithreading and Communication”, Journal of Parallel and Distributed Computing, 37, pp.70-82, 1996

[13] I.Foster, C. Kesselman, S. Tuecke, “The Nexus Task-Parallel Runtime System”, Proc. 1st Int'l Workshop on Parallel Processing, pp. 457-462, 1994

[14] T. C. Lee, C. Kesselman, J. Stepanek, R. Lindell, S. Hwang, B. Scott Michel, J. Bannister, I. Foster, A. Roy, “The Quality of Service Component for the Globus Metacomputing System” , Proc. IWQoS '98, pp. 140-142, 1998.

[15] I. Foster, N. Karonis, “A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems”, Proc. SuperComputing 98

[16] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke, “A Directory Service for Configuring High-Performance Distributed Computations”, Proc. 6th IEEE Symp. on High-Performance Distributed Computing, pp. 365-375, 1997

[17] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke, “A Security Architecture for Computational Grids”, Proc. 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92, 1998.

[18] I. Foster, N. T. Karonis, C. Kesselman, S. Tuecke, “Managing Security in High-Performance Distributed Computing”, Cluster Computing 1(1) pp. 95-107, 1998.

[19] I. Foster, N. Karonis, C. Kesselman, G. Koenig, S. Tuecke, “A Secure Communications Infrastructure for High-Performance Distributed Computing”, 6th IEEE Symp. on High-Performance Distributed Computing, pp. 125-136, 1997

[20] P. Stelling, I. Foster, C. Kesselman, C.Lee, G. von Laszewski, “A Fault Detection Service for Wide Area Distributed Computations”, Proc. 7th IEEE Symp. on High Performance Distributed Computing, to appear, 1998.

[21] J. Bester, I. Foster, C. Kesselman, J. Tedesco, S. Tuecke, “GASS: A Data Movement and Access Service for Wide Area Computing Systems”, Sixth Workshop on I/ O in Parallel and Distributed Systems, May 5, 1999

[22] I. Foster, D. Kohr, R. Krishnaiyer, J. Mogill, “Remote I/O: Fast Access to Distant Storage”, Proc. Workshop on I/O in Parallel and Distributed Systems (IOPADS), pp. 14-25, 1997

[23] Bruce Schneier, Applied Cryptography, 2nd edition. John Wiley & Sons, 1995.

[24] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, Handbook of Applied Cryptography**,** CRC Press, 1996

CORBA: A PRACTICAL INTRODUCTION

Serguei Kolos [1])
CERN, Geneva, Switzerland

Abstract

Common Object Request Broker Architecture (CORBA) is the recent answer for the need for interoperability in the modern distributed computing environment. This architecture allows objects to communicate with one another regardless of their location and implementation. CORBA hides the differences between the operating systems, programming languages and address spaces for the objects implementations.

This text will give an overview of the CORBA along with the Java programming example that illustrates the process of the CORBA based distributed system development. At the end some examples of how CORBA is currently being used in HEP experiments will be given.

1. INTRODUCTION

Throughout the history of computing the demand for distributed computation has been growing permanently. The evolution in this area has been always moving to the direction of increasing level of abstraction for the communication implementation. The first major step on this road was so-called *socket*[1] interface that unifies the way in which a low-level protocol have been used. It is a relatively simple set of generic procedures that allow bidirectional data exchange between applications regardless of their location. The applications could be executing either on the same or on different computers even if those computers run different operating systems.

The Remote Procedure Call (RPC)[2] layer brings into use the idea of control messages exchange instead of just data exchange between distributed applications. RPC gives the possibility to unify local and remote procedures invocations at the level of programming language code. The remote procedures could be declared and called exactly in the same way as the local ones with the only omission that the invocation errors shall be handled differently.

Another step has been done by the Distributed Computing Environment (DCE)[3] system. DCE is a product of the Open Software Foundation (OSF)[4] that puts into practice the idea to use a special language for the declaration of communication interfaces. The Interface Definition Language (IDL) has been promoted to provide a service interface description. The IDL declaration is the only information required for the development of both a service provider and a service requestor for the interface it describes. The important result of the IDL innovation is the ability to use different programming languages for the implementation of the applications that are taking part in communications.

Each round in this evolution was another step on the road of unification of the local and distributed computing models. The resent step in this evolution was the introduction of the CORBA standard by the Object Management Group (OMG)[5] in 1991. The major innovation of the OMG that quickly made CORBA the leading communication standard was the application of the Object technology to the communication domain. CORBA puts into practice the Object Oriented Design (OOD) methods for software engineering. OOD as a method of modelling a problem by taking a balanced view about objects and the operations performed upon them, was proposed by Grady Booch[6]. The classical OOD model does not take care of the objects' implementation. It defines the object interfaces and relationships regardless of where the objects are and how they have been implemented. But for a number of years this model had been rather a theoretical abstraction because of

---

1. On leave from the Petersburg Nuclear Physics Institute, Gatchina, Russia

the essential gap between the OOD model and implementation technologies. The CORBA standard is a great contribution to the applicability of the OOD technology to a real computing problems. CORBA acts as a bridge between the object design methods and objects' implementation. The Object-Oriented IDL perfectly maps to the OOD model allowing to express in a formal way any OOD paradigm. Most of the modern OOD tools like Rose[7], StP[8], Together[9], etc. are able to generate automatically the interfaces in OMG Interface Definition Language (IDL)[1] from the design artifacts. This IDL definition is substantial for the implementation of the objects that can be used either locally or remotely.

The next two chapters give the overview of the CORBA standard and the OMG reference architecture. An example that shows how CORBA can be used for the implementation of the system has been designed using OOD methods will be shown in the next chapter. At the last chapter some references to the applications of CORBA in HEP experiments are given.

2. CORBA OVERVIEW

CORBA standard is based on the classical object model[10] that defines two kinds of distinct entities: classes that support encapsulation, inheritance and polymorphism and objects that are classes' instances. The fundamental principle that was recognized by CORBA is the independence of the behavior of an object from its implementation. The behavior of an object is defined by it's interface, where the interface is a set of object method signatures. From this point of view the CORBA standard can be seen as consisting of two logical levels:

- Interface definition level: Defines syntax and semantics for the objects' interface description by means of OMG IDL[11]. The way in which IDL interfaces can be mapped to a programming language is standardized by the CORBA Language Mappings[12]. The Language Mapping specifications provide a link to the next CORBA level.
- Communication media level: Specifies how communications have to be implemented. For this purpose the Object Request Broker (ORB) specification is introduced. The ORB acts as an inter-object communication bus providing a set of interfaces for object creation, registration and access control.

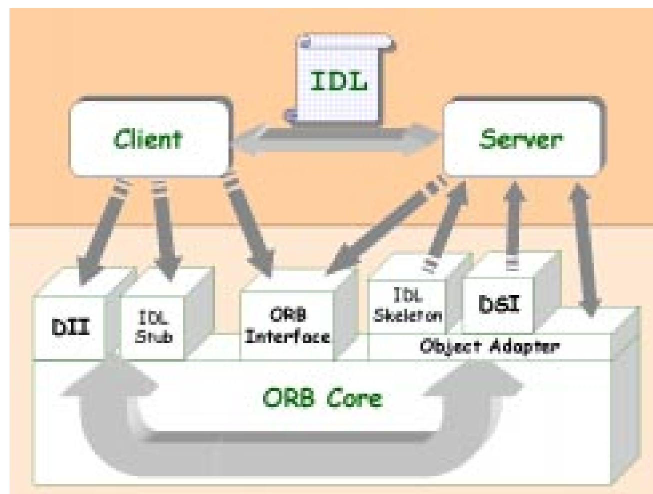


Fig. 1 The structure of the CORBA standard

Figure 1 shows the structure of the CORBA standard. Different components of the ORB are briefly described in the next paragraphs. Those who are interested in a more detailed description we can address to the respective chapters of the [13].

1. Do not confuse it with the DCE IDL. The OMG IDL is essentially different from the DCE's one.

2.1 Object Interface Definition

The basis for communication description is formed by the OMG IDL that was the first component defined by the CORBA standard. The OMG IDL is widely accepted as the *de facto* standard for objects' interface description. It is often used as a general purpose interface description tool even for a description of a non distributed objects interfaces. There are two fundamental features of the OMG IDL that make it so popular:

- OMG IDL is able to provide a comprehensive object interface description using intuitive self-explained semantics. A service description, has been done on OMG IDL, provides all the necessary information for the independent development of both a service provider and a service consumer.
- Any IDL construct can be easily mapped to most of the existing programming languages.

These facilities make the OMG IDL extremely useful for the OOD. It is IDL that bridges object design and object implementation phases of the software life cycle. This chapter gives an overview of the IDL and explains what the CORBA Language Mappings are.

2.1.1 OMG Interface Definition Language

As it was already mentioned an IDL declaration looks very self-explained and understandable because the OMG IDL uses the same grammar and lexical rules as C++. Most of the IDL's keywords are well known for the software developers and ordinarily it takes them just a few hours to learn IDL.

The key concept of an IDL declaration is an interface. Interface declaration consists of an interface header and interface body. Interface header consists of the optional keyword '*abstract*', the interface name and optional inheritance specification. An interface body contains attributes and operations declarations. Figure 2 shows an example of the IDL interfaces declaration.

```
1:   abstract interface container {
2:      long size();
3:   };
4:
5:   interface iterator: container {
6:      any next();
7:   }
8:
9:   interface list: container {
10:     void add(in any elem);
11:     iterator create_iterator();
12:  };
```

Fig. 2 IDL interfaces declaration

An interface can be derived from another interface, which is then called a base interface. A derived interface can declare it's own attributes and operations. Multiple inheritance is allowed - an interface may be derived from any number of base interfaces. The order of derivation is not significant.

Data can be defined only as attributes of the interfaces. The attribute declaration means that a respective accessor an mutuator methods will be available for this attribute. If an attributed is prefixed with the read-only keyword only the accessor method will be available for the specific attribute. Thus the attribute construct does not declare a storage for the data value but instead just defines a data access interface. The IDL declarations shown in Table 1 are equivalent.

Table 1 Two alternative ways of interface description

1: interface person { 2: 3: attribute octet age; 4: 5: read-only attribute string name; 6: };	1: interface person { 2: void set_age(in octet ag); 3: octet get_age(); 4: 5: string get_name(); 6: };

The OMG IDL specification define a number of basic types that can be used for the attributes and method parameters declarations. These types covers all the possible basic computational units in programming languages. They are shown in Table 2.

Table 2 Basic OMG IDL types

OMG IDL type	Matched by
char, wchar	simple and wide character literals
octet	unsigned 1-byte value
short, unsigned short	signed and unsigned 2-byte values
long, unsigned long	signed and unsigned 4-byte values
long long, unsigned long long	signed and unsigned 8-byte values
float, double	simple and double precision floating point values
string, wstring	sequences of simple and wide character literals
enum	set of user defined values

These basic types can be used to construct complex ones by utilizing the structure, union and sequence patterns. Figure 3 shows an example of the complex types definition.

```
 1:   enum PrimitiveKind{
 2:      pk_unknown, pk_char, pk_octet, pk_short, pk_ushort,
 3:      pk_long, pk_ulong, pk_float, pk_double, pk_boolean, pk_string,
 4:   };
 5:
 6:   union DataEntry switch(PrimitiveKind){
 7:      case pk_char:          char            de_char;
 8:      case pk_octet:         octet           de_uchar;
 9:      case pk_short:         short           de_short;
10:      case pk_ushort:        unsigned short de_ushort;
11:      case pk_long:          long            de_long;
12:      case pk_ulong:         unsigned long   de_ulong;
13:      case pk_float:         float           de_float;
14:      case pk_double:        double          de_double;
15:      case pk_boolean:       boolean         de_bool;
16:      case pk_string:        string          de_string;
17:   };
18:
19:   struct NamedDataEntry{
20:      string      name;
21:      DataEntry   entry;
22:   };
23:
24:   typedef sequence<DataEntry> Data;
25:
26:   typedef sequence<NamedDataEntry> NamedData;
```

Fig. 3 OMG IDL complex types

There are some other constructs in IDL that have not been mentioned yet. They are *exceptions* that are used to indicate the exceptional conditions of the methods execution, the *oneway* keyword used to declare the asynchronous style of the method invocation, the *module* that is the namespace

declaration that prevents the names declared in the scope of this module from clashes with other IDL declarations, etc. For the complete specification of the IDL syntax and semantics see [11].

OMG IDL is a permanently evolving standard that tracks the major innovations in the OOD world while keeping the backward compatibility with the previous versions of the specification. It give us a good reason to be optimistic for the future of the OMG IDL as a basic method of objects' interface description.

2.1.2 Languages Mappings

IDL is purely declarative language. It is used to declare interfaces and can not be used for their implementation. IDL is used to express the OOD patterns in a formal way but it is still an abstraction that requires another type of formal definition that specifies how IDL interfaces can be mapped to the 'real world' of software implementation. Such specification is provided by the CORBA standard by means of Language Mappings for several programming languages.

A Language Mapping defines a programming language counterpart for each construct of the OMG IDL. This definition allows to generate programming code from the IDL declaration automatically. This programming code is an interface declaration that is equivalent to the IDL one but is expressed by means of a programming language.

For various languages these mappings are essentially different. For example *interface* declaration corresponds to *class* in C++, *interface* in Java and *structure* in C language. All the conformity between the IDL and a programming language are formally described by the standard in such a way that source code compatibility between different ORBs will very probably be possible in the nearest future. Table 3 shows the Java mappings for the IDL types accordingly to the OMG specification[14].

Table 3 Java mappings for OMG IDL types

OMG IDL type	Java mapping types
short, long, long long	short, int, long
unsigned short, unsigned long, unsigned long long	unsigned short, unsigned int, unsigned long
float, double	float, double
char, boolean, octet	char, boolean, byte
any	org.omg.CORBA.Any class
string, wstring	string
struct, union, enum, exception	class
sequence<type>	type[]
interface	interface
module	package

Currently, the CORBA standard defines mappings for the following languages: C, C++, Smalltalk, COBOL, Ada, Java and Lisp. Several independent companies develop their own ORBs with the mappings to languages that are not part of the CORBA standard. For example Xerox[15] has an ORB called Inter Language Unification (ILU)[16] with mappings to the Python and Modula2 languages. There are no limitations for the use of such language mappings for the CORBA object development. The CORBA client implemented in Lisp can call the methods of a CORBA server that

have been written in any of the 'standard' languages and vice versa. The only trouble with a non-standard mappings is a possible source code incompatibility between different ORB implementations.

2.2 Object Request Broker architecture

As it was already mentioned a programming language code can be automatically generated from the IDL declaration. This task is done by a special application called an IDL translator or an IDL compiler. The IDL compiler takes an IDL statements as input and produces a code for a specific programming language according to the OMG language mapping specification. Thus the IDL compiler is a glue that links together two CORBA levels: an abstract interface definition and a concrete ORB connection implementation.

2.2.1 Stubs and Skeletons

The target code generated by the IDL translator appears in pairs: client-side and server-side. The client-side mapping is called *stub* or *proxy* - it is a mechanism that creates and issues requests from a client. The server-side code is called *skeleton* - it is a mechanism that delivers requests to CORBA Object implementations. Such code is statically bound with the respective server or client code at compile and link time. Figure 4 shows the workflow for CORBA application development.

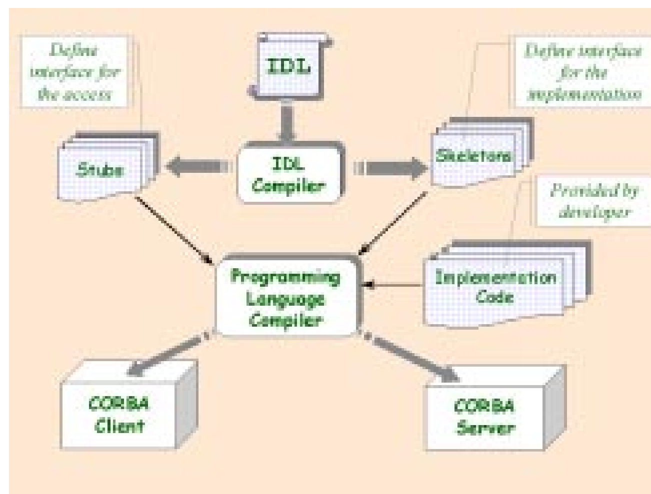


Fig. 4 Workflow for CORBA application development

Before compilation the skeleton code needs to be fleshed out with the actual implementation code for each method. Stubs are basically complete. The *stub* methods can be used directly to request a service described by the IDL.

When a *stub*'s method is called by a client application it marshals a request to the ORB Core doing a conversion of a request from the programming language representation to one that is suitable for transmission over the connection to the target object. Then the ORB Core is responsible for the request transportation to the server that holds the target object and passing this request to the *skeleton* code. The *skeleton* unmarshals the request doing a conversion to a programming language, that is not necessarily the same as for the client application, and dispatches the request to the appropriate object. Dispatching through stubs and skeletons are often called *static invocation*. It is shown in Figure 5.

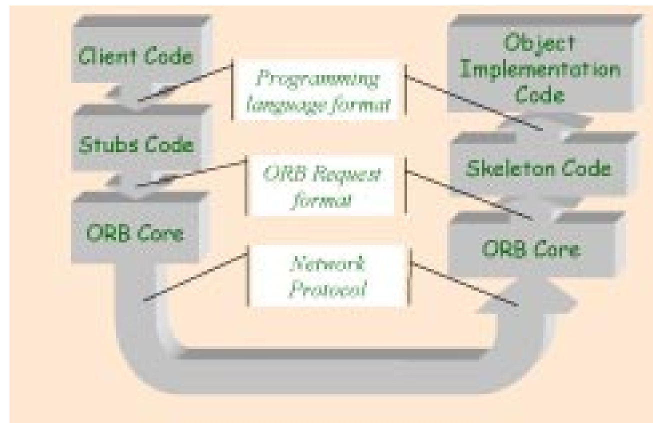


Fig. 5 Static method invocation

It is necessary to mention that the ORB Core that does the request transportation can be represented in practice by the combination of the two Core components belonging to different ORBs. In other words a server and a client application can be implemented using different ORBs.

2.2.2 Dynamic Invocation

In addition to static invocation via stubs and skeletons, CORBA supports two interfaces for dynamic invocation:

- Dynamic Invocation Interface (DII)[17] supports dynamic client request invocation.
- Dynamic Skeleton Interface (DSI)[18] provides dynamic request dispatching to the implementation objects.

DII and DSI can be viewed as a generic stub and generic skeleton respectively. These interfaces are provided by the ORB and are independent from IDL interfaces of the objects being invoked. The main purpose of these generic interfaces is to support the implementation of the bridges between CORBA and non-CORBA communication systems.

Using DII a client application can invoke requests without having compile-time knowledge of the object's interfaces. A request consists of an object reference, an operation name and a list of parameters. Each parameter has a name and a value. Parameters' order is essential and must confirm to the order in which they are defined for the interface. Figure 6 shows how the generic bridge can be implemented using DII.

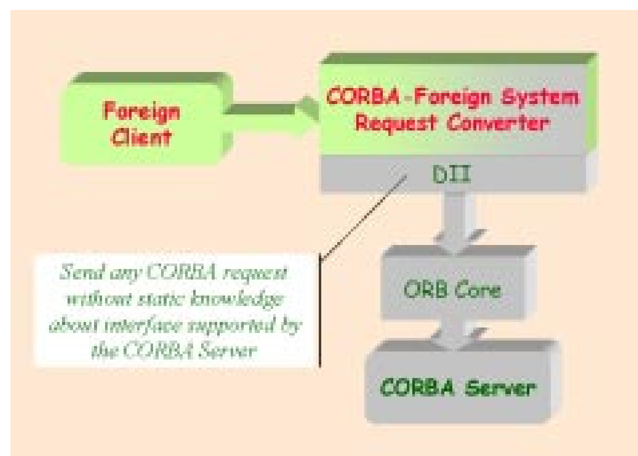


Fig. 6 The DII based bridge between an ORB and a non-CORBA system

The counterpart for the DII is DSI. DSI allows servers to be implemented without having skeletons for the objects compiled statically into the program. This concept was introduced in CORBA 2.0 as a possible mean of interoperability implementation. The DSI based bridge architecture is presented in Figure 7.

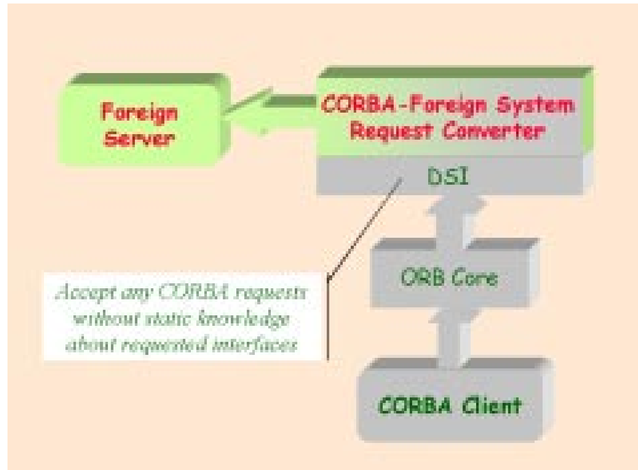


Fig. 7 The DSI based bridge between an ORB and a non-CORBA system

2.2.3 ORB Core

The main component of an Object Request Broker is called the ORB Core. It serves as a communication bus for the request transportation. The ORB Core assures the transparency of the following communication aspects:

- Object location: Mutual location of the server and client objects is transparent on the level of implementation code. The objects can be parts of one program instance, sharing runtime support in one memory image; they can be parts running in different program instances on different machines connected either with a local or global network.
- Communication protocol: The ORB has to use the same communication protocol (e.g. TCP/IP, UDP, RPC over TCP/IP, etc.) on both client and server ends of connection. But it is completely transparent for the programming objects which communication mechanism is actually used. This mechanism can be swapped with another one without affecting the objects' implementation.

2.2.4 Object Adapter

The Object Adapter provides an intermediate layer between the object implementation and the ORB Core. The OA is responsible for the following operations:

- Object reference handling: Object references are created in servers. Once they have been created, they may be exported to clients. From this model's perspective, object references encapsulate object identity information and information required by the ORB to identify and locate the server and OA with which the object is associated.
- Object request transportation: When a client issues a request, the ORB first locates an appropriate server and then locates the appropriate OA within that server. Once the ORB has found the appropriate OA, it delivers the request to that OA. Then the OA invokes the appropriate method on the request's target object.

The first specification for the object adapter provided by OMG was the Basic Object Adapter (BOA). But very soon it was found that some important BOA operations that had not been clearly defined by the standard were implemented very differently by various ORB vendors. The main differences have been the object registration and object activation operations which result in nontrivial portability problems between different ORBs.

OMG recognized these problems and a new Portable Object Adapter (POA)[19] specification was established in the recent CORBA versions. POA addresses the following issues that have been missed or not fully addressed by the BOA specification:

- Object portability: Allow programmers to construct object implementations that are portable between different ORB products.
- Object persistence: Provide support for objects whose lifetimes span multiple server lifetimes.
- Object activation: Provide support for transparent activation of objects.

2.2.5 ORB Interface

The ORB Interface[20] defines the application program interface for the operations implemented by the ORB. These operations are the same among all CORBA brokers and do not depend on the particular object adapter used. The main responsibility of the ORB Interface is to provide a portable means by which service implementation objects can be accessed by clients willing to utilize these services. The ORB Interface defines such access mechanism by the means of object reference handling. An object reference may be translated into a string by the operation *object_to_string*. The string value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the *string_to_object* operation will accept a string produced by object_to_string and returns the corresponding object reference. The string format must be recognized by any ORB implementation. Figure 8 shows how a client application can establish a connection to a servant using the ORB Interface methods described above.

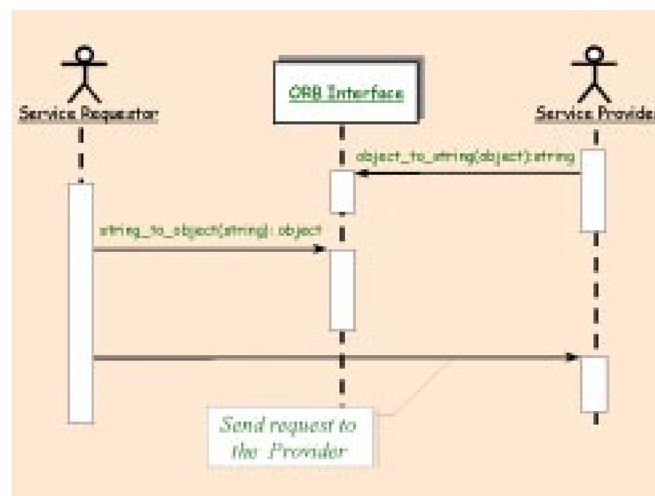


Fig. 8 CORBA objects connectivity Sequence diagram (UML notation)

2.3 Interoperability

The original request for interoperability was issued by the OMG in 1993. It defined interoperability as an ability of a client on ORB 'A' to invoke an OMG IDL defined operation on an object on ORB 'B'. It is assumed that CORBA brokers 'A' and 'B' have been developed independently. In the recent CORBA specifications the idea of interoperability is defined as "Interoperability is a comprehensive flexible approach to support network of objects implemented and managed by multiple heterogeneous CORBA-compliant ORBs"[21].

The interoperability architecture defines two concepts to achieve interoperability: mediate and immediate bridging of ORBs. Let us assume that any ORB implementation or any other communication standard forms its own domain of communication. For the mediate bridges, elements of one domain are transformed from the format internal to this domain to another format that is agreed to be common for all domains participating in interactions. For immediate bridging, elements of the

interaction are transformed directly from the internal format of one domain to the internal format of the other.

The Generic Inter-ORB Protocol (GIOP)[22] can be seen as the common basis for the mediated bridging approach implementation. The protocol specification is the common agreed format that is recognized by any interoperable ORB. GIOP specifies a standard transfer syntax and a set of message formats for communications between ORBs. The GIOP protocol is simple, scalable and relatively easy to implement.

The Internet Inter-ORB Protocol (IIOP)[22] specifies how the GIOP messages are exchanged through TCP/IPC connections. IIOP can be seen as a mapping of GIOP for a specific transport. The IIOP is the standard protocol that is supported by almost any of the current ORBs as the default one. Thus, practically, most of the existing CORBA brokers are interoperable on the level of IIOP and are able to communicate with one another.

An example of an immediate bridge is a bridge between a CORBA and non-CORBA system for which there is no common intermediate message exchange format. The Figures 6 and 7 shown the examples of such an approach.

An ORB is considered to be fully interoperability-compliant when it supports both the IIOP protocol and standard CORBA interfaces such as ORB Interface, DSI and DII.

3. OBJECT MANAGEMENT ARCHITECTURE

While the modern computing paradigm tends to the distributed computing, the distributed software products becomes more and more complex and the software life circle issues becomes more and more important. The critical parameters for the modern software are: the time to develop it, the ability to maintain and enhance it and the time it takes to learn to use it. The Object Management Group provides a reference architecture that is called Object Management Architecture (OMA)[23] in order to address these issues. The OMA defines a common framework that is intended to simplify the information systems development and support via the definition of the joint public services based on the common standard. The communications heart of the OMA is Object Request Broker component. As it is shown on Figure 9 the ORB joins three main OMA components:

- Applications Objects: These are the CORBA objects implemented by independent developers intended to fulfill their specific needs. These objects can be reused by the other developers and they might become a candidates for the OMG standardization.
- Object Services: The Object Services standardize the life cycle management of the distributed objects. They will be explained in more details in the following section.
- Common Facilities: They provides a set of generic application functions that can be configured to the requirements of a specific configuration. The facilities already formalized by the OMG are Internationalization, Time and Mobile Agent [24].

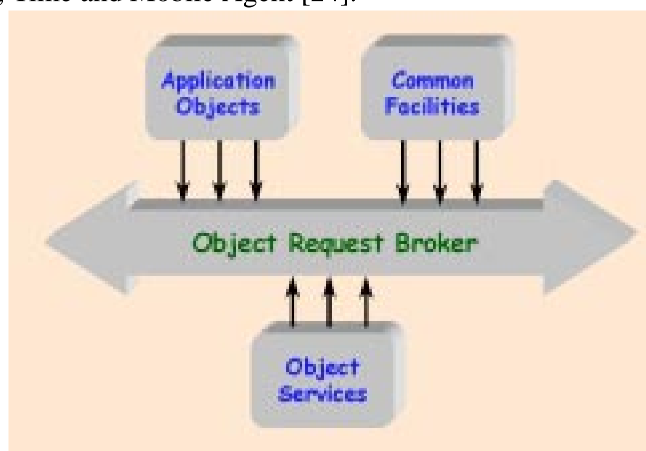


Fig. 9 Object Management Architecture

3.1 OMG Services

The Object Services cover different aspects of the object management including the objects creation, control access to objects, objects relocations and the objects relationship maintenance. The Object Service components provide the generic environment in which single objects can perform their tasks. Standardization of Object Services leads to consistency over different applications and improved productivity for the developer. Specifications for the Object Services that have been adopted as standards by the OMG are contained in [25]. There are sixteen service specifications provided by the OMG at the moment. The first service for which the definition has been provided and the most often used one is the Naming Service that will be explained in a more details below in order to give an impression of a CORBA Service essence to the reader.

3.1.1 Naming Service

The Naming Service[26] specification defines a federated (hierarchical) naming service that is commonly used to allow to use a human-readable format for the programming objects references. It does this by providing the name-to-object associations from which any object can be uniquely identified by the associated name.

A name-to-object association is called a name binding. A name binding is always defined relative to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. To resolve a name is to determine the object associated with the name in a given context. To bind a name is to create a name binding in a given context. A name is always resolved relative to a context, there are no absolute names. Because a context is like any other object, it can also be bound to a name in a naming context. Figure 10 shows how the Naming Service can be used for the objects registration and access control. One can notice that this figure is very similar to the Figure 8 that shows how object connection can be established via the ORB Interface facility. The Naming Service methods are intended to be used instead of the *string_to_object* and *object_to_string* pair.

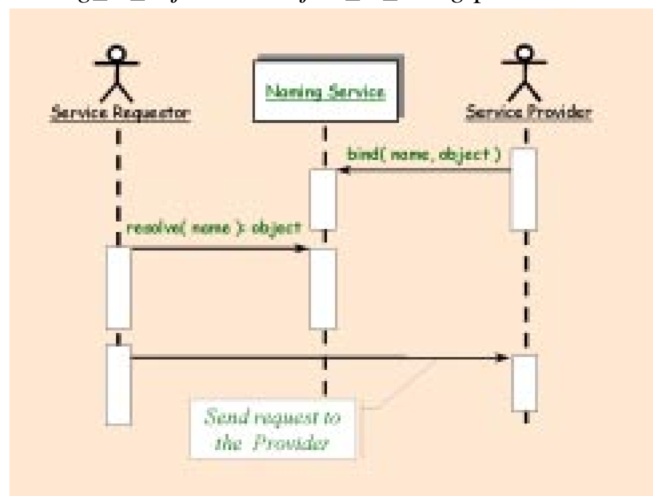


Fig. 10 Naming Service Sequence diagram (UML notation)

The CosNaming Module is a collection of interfaces that together define the naming service. This module is described in OMG IDL and contains two interfaces:

- The NamingContext interface - allows objects bindings and names resolution;
- The BindingIterator interface - allows to iterate through the bindings.

4. DEVELOPING A DISTRIBUTED APPLICATION WITH CORBA

In this chapter a comprehensive example of CORBA based distributed system design and development will be given. For the design and implementation described in this chapter the following software development tools have been used:

- Rational Rose[7] framework has been used for the diagram drawing and classes definition. Rose is an analysis and design framework that enables business analysts and software developers to specify business models and software applications graphically.
- Two ORBs have been used for the development: the JavaIDL[27] that is a Sun Microsystems Java ORB included to the JDK 1.2[28] and ORBacus 3.1.3[29] that is a C++/Java ORB of Object Oriented Concept[30].
- The *idltojava*[31] compiler version 1.2 that is the IDL compiler of Sun Microsystems has been used to generate Java *stub* code.
- The *idl* compiler that is the part of ORBacus 3.1.3 distribution has been used to generate C++ *skeleton* code.

4.1 Problem definition and proposed solution

High energy physics experiments investigate reactions between colliding elementary particles. To this purpose data on the particles leaving the collision point are recorded in large detectors and stored in digital form. The set of data recorded per collision is called an event. The events are the basic units for further investigations, which are done by powerful pattern recognition and analysis programs. One of the approaches used for the physical data estimation is a visual analysis of single events. For the event visualization a special class of application called Event Display is used. An Event Display provides independent method for the estimation of the information relevance by visualizing the event taken from the event storage system.

One of the important issues for the Event Display is the possibility to run it remotely, i.e. on a computer that does not have direct access to the event storage system. The possible solution might be to run the Event Display application on the machine that holds the event storage system while redirecting it's output to the user's machine via the standard X server display redirection facility. The critical point here is the network performance that sometimes is not enough to use this approach. As a solution we propose to separate out Event Display into two subtasks:

- Event Painter task that is the application that retrieves event information from the data storage and paints the event image in memory, but does not display it. This application is running on the machine with a direct data storage access. This Event Painter is in fact the classical Event Display application with the only difference that the Painter does not display the event image it has prepared.
- Event Visualizer is the task that retrieves the image from the Event Painter task via a CORBA interface and display this image on a user machine. This application is running on a remote machine that can not access the event storage system directly.

The Event Visualizer identifies an event it is willing to present. This identification can be done by passing the run and event numbers to the Event Painter. The Painter prepares an image in memory and passes it back in the machine independent format. For this simple example one of the well known graphics formats can be used here, for example JPEG, GIF, PNG, etc. The most suitable one in fact is a GIF format because it supports a suitable data compression without major loss of the image quality and can be easily displayed by the standard means of Java language. Figure 11 shows the information exchange between these tasks.

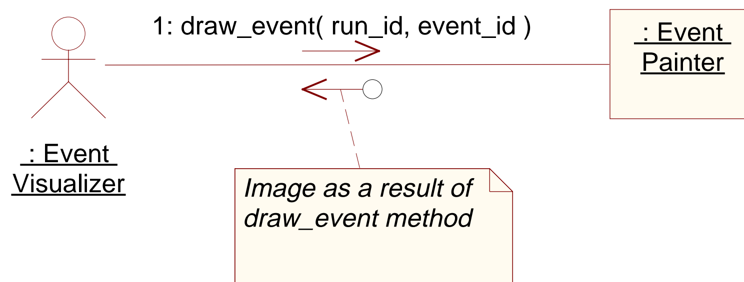


Fig. 11 Event Display Collaboration diagram (UML notation)

In reality the complexity of the modern detectors and events has increased so significant that simple non-interactive event image is not always suitable for the visual analysis. The modern Event Display is an interactive tool that can perform different image transformations like zoom, rotation, etc. upon a user request. For such facility it is necessary to use another image description format, for example XML[32].

Use of CORBA for this system implementation offers the important advantage: the most suitable programming languages can be chosen for the Painter and Visualizer implementation. It seems that visualization task can be better implemented on Java but the Painter may require a different language that is defined by the event storage system API. Most of the event storages that are widely used in physics experiments are not accessible from Java and provides ordinarily a Fortran or C/C++ interfaces. So that it worth using a C++ for the Event Painter implementation.

Another significant advantage of the proposed approach is independence of the Event Visualizer from the physical nature of events and detector geometry. Different experiments require different visualization technics for the experimental data representation. Therefor different implementations of the Event Painter must be provided but the same Event Visualizer can be used for all of them.

The definition of the CORBA interface for the Event Painter task and implementations for the both event analysis and event visualization tasks are discussed below.

4.2 Events access use cases

As it was discussed in the previous section the Event Painter interface is able to draw an event that is identified by the run and event numbers supplied by the Event Visualizer. But the Event Visualizer must have a way to find these numbers because it can not retrieve them from data storage itself since it has no access to it. The simplest way is to add to the Event Painter interface the methods that return the lists of run and event numbers. Thus the first operation to be done by the Event Visualizer is a request for the list of valid run numbers. Then for any run number it asks for a list of valid event numbers. This operation can result in the *BadRunNumber* exception if the wrong run number has been provided. In the opposite case the event drawing operation can be requested. The possible exceptions for this operation are: *BadRunNumber* and *BadEventNumber*. Upon a successful completion the event image can be displayed. Figure 12 shows the use cases described above.

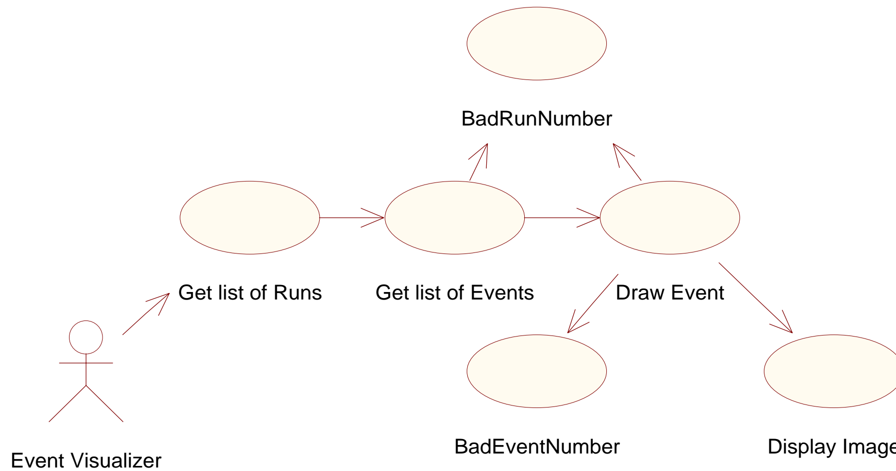


Fig. 12 Event Display Use-Case diagram (UML notation)

4.3 Classes definition

The next step towards the implementation is the definition of the classes that should be capable of handling the use cases described above. The interface called Painter and four data types have been defined for this purpose. The Painter interface declares the methods which cover all the aspects of the interface required by the Event Visualizer task. The data types support the return values and possible exceptions for the Painter's methods. These classes are shown on the Figure 13.

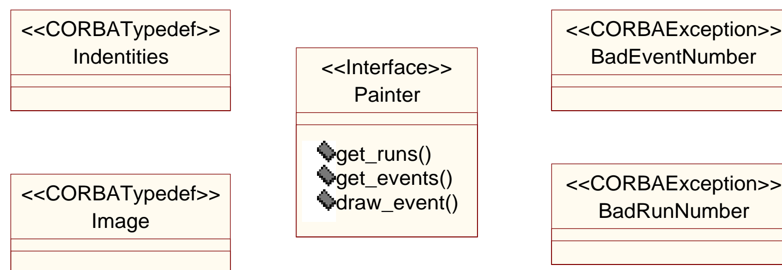


Fig. 13 Event Display Class diagram (UML notation)

The *Identities* is a type name for the sequence of the long integer values. It is used to specify a result for the *get_runs* and *get_events* methods. The *Image* type is defined as a sequence of bytes and is intended to be a return type of the *draw_event* method. The *BadEventNumber* and *BadRunNumber* are utilized to indicate the exceptional conditions for the Painter's methods execution. *BadEventNumber* can be raised by the *draw_event* method and *BadRunNumber* by both *get_events* and *draw_events* methods. All this information has been attached to the respective classes with the help of the Rose class diagram editor. This information is required to support the automatic IDL file generation.

4.4 OMG IDL declaration

Figure 14 shows the IDL declaration generated by Rose from the class diagram shown earlier.

```
1:   module Event {
2:      typedef sequence<octet> Image;
3:      typedef sequence<long> Identities;
4:
5:      exception BadRunNumber {};
6:      exception BadEventNumber {};
```

```
7:
8:      interface Painter {
9:         Identities get_runs();
10:        Identities get_events(in long run_id) raises (BadRunNumber);
11:        Image draw_event(in long run, in long event, in long width, in long
                              height) raises (BadRunNumber, BadEventNumber);
12:     };
13:  };
```

Fig. 14 Event Display interface (IDL)

It is necessary to run a specific IDL compiler to produce a *stub* and *skeleton* code from this IDL declaration. Since it was decided to use different languages for the Event Painter and Event Visualizer implementation different IDL compilers have to be used. The *idltojava* of Sun provides a Java *stubs* for the Event Visualizer and *idl* compiler by OOC has been used to generate a C++ *skeleton* for the Event Painter.

4.5 Event Painter implementation

4.5.1 Event Painter interface implementation

Figure 15 shows how to declare the C++ implementation class for the Event Painter interface. The Event_Painter_skel class that is a descendant of the Event_Painter_impl has been generated by the IDL compiler.

```
1:   class Event_Painter_impl: public Event_Painter_skel
2:   {
3:   public:
4:
5:      virtual Event_Identities* get_runs() {};
6:      virtual Event_Identities* get_events(CORBA_Long run_number) {};
7:      virtual Event_Image* draw_event(CORBA_Long run,
8:                                      CORBA_Long event,
9:                                      CORBA_Long width,
10:                                     CORBA_Long height);
11:  };
```

Fig. 15 Declaration of the class that implements Event Painter (C++ language)

The virtual methods declared in the *Event_Painter_impl* class are inherited from the *Event_Painter_skel* class where they are defined as pure virtual methods. In order to provide an implementation for the Event Painter it is necessary to implement all these methods. Figure 16 shows how this implementation can be done. The functions *next_run* in line 8 and the *next_event* in line 21 represent a virtual API to the event storage system. For the actual implementation they shall be replaced with the adequate real data storage API calls.

```
1:   Event_Identities * Event_Painter_impl::get_runs()
2:   {
3:   // create the result sequence
4:      Event_Identities list = new Event_Identities;
5:
6:   // retrieve run numbers from the event storage system
7:      unsigned long run_number;
8:      while((run_number = next_run())!= -1)
9:         list.add(run_number);
10:
11:     return list;
12:  };
13:
14:  Event_Identities * Event_Painter_impl::get_events(CORBA_Long run_number)
15:  {
16:  // create the result sequence
```

```
17:     Event_Identities list = new Event_Identities;
18:
19:  // retrieve event numbers from the event storage system
20:     unsigned long event_number;
21:     while((event_number = next_event(run_number))!= -1)
22:        list.add(event_number);
23:
24:     return list;
25:  };
26:
27:  Event_Image * Event_Painter_impl::draw_event(
28:        CORBA_Long run, CORBA_Long event, CORBA_Long width, CORBA_Long height
29:  )
30:  {
31:     unsigned char * image_bytes;
32:     unsigned long image_size;
33:  // prepare the image in memory
34:     .....
35:  // create a bytes sequence to pass back to the Visualizer
36:     Event_Image * image = new Event_Image(image_size, image_size, image_bytes,
                                  true);
37:     return image;
38:  }
```

Fig. 16 Event Painter interface implementation (C++ language)

4.5.2 Event Painter task implementation

Since we have implemented the Event Painter interface there is only one thing that remains to be done - the EventPainter_impl class instance has to be created and registered with the ORB. Figure 17 shows how this can be done.

```
1:    int main(int argc, char* argv[], char*[])
2:    {
3:        try
4:       {
5:       // Create ORB and BOA
6:          CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
7:          CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
8:
9:       // Create implementation object
10:         Event_Painter_var p = new Event_Painter_impl();
11:
12:      // Print stringified object reference to the standard output
13:         CORBA_String_var s = orb -> object_to_string(p);
14:         cout << s << endl << flush;
15:
16:      // Run implementation
17:         boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
18:      }
19:      catch(CORBA_SystemException& ex)
20:      {
21:         OBPrintException(ex);
22:         return 1;
23:      }
24:      return 0;
25:   }
26:
```

Fig. 17 Event Painter task implementation

The ORB initialization is done in line 6. The *CORBA_ORB_init* method creates an instance of the CORBA_ORB class that encapsulates all the methods of the ORB Interface that has been described in section 2.2.5. In the next line the instance of the Basic Object Adapter is created. The servant registration with the BOA instance is done implicitly during the construction of the *Event_Painter_impl* object. The BOA instance is used later in line 17 to call *impl_is_ready* method that is responsible for the acceptance of the external requests. The ORB instance is used to call the *object_to_string* method to convert the Event Painter object reference to string. This string is printed to the standard output stream and is intended to be used by the Event Visualizer for the access to the Event Painter.

4.6 Event Visualizer implementation

In order to deal with the interface defined for the Event Painter no additional code is required. The simple Java client application that uses directly the classes generated by the IDL compiler is shown in Figure 18. This code contains just the essential information and does not show all the aspects of the Java interface creation.

```
1:   public class EventVisualizer extends JFrame{
2:      static org.omg.CORBA.ORB orb;
3:      static JLabel label;
4:
5:      public EventVisualizer () {
6:      // create all the necessary graphical components here
7:         ....
8:      // this Jlabel will be used to display event image
9:         getContentPane().add(label = new JLabel());
10:     }
11:
12:     public static void main(String args[]){
13:     // create and show application's main frame
14:        EventVisualizer client = new EventVisualizer();
15:        client.show();
16:
17:     // initialize ORB
18:        orb = org.omg.CORBA.ORB.init((String[])null, null);
19:     // convert parameter string to the Event Painter reference
20:        org.omg.CORBA.Object obj = orb.string_to_object(args[0]);
21:        Event.Painter ed = Event.PainterHelper.narrow(obj);
22:
23:     // try to draw an event
24:        try{
25:           // call event_draw method for the event 'm' of the run 'n'
26:           byte[] data = ed.draw_event(run_id, event_id, frame.getWidth(),
                                 frame.getHeight());
27:        // display image (GIF and JPEG images can be displayed)
28:           label.setIcon(new ImageIcon(data));
29:        }
30:        // catch bad run number exception
31:        catch(Event.BadRunNumber ex){
32:           System.err.println("Bad Run number is used.");
33:        }
34:        // catch bad event number exception
35:        catch(Event.BadEventNumber ex){
36:           System.err.println("Bad Event number is used.");
37:        }
38:     }
39:  }
40:
```

Fig. 18 Event Visualizer implementation (Java)

The call to the *init* method of the org.omg.CORBA.ORB class in line 18 returns the ORB instance that can be used to call the *string_to_object* method. This method takes the first command line argument passed to the Visualizer application and tries to convert it to the Event Painter object reference. The *string_to_object* method always returns a reference to a generic CORBA object so it is necessary to cast this reference to a specific type that is *Event.Painter* in this case. It is done in line 21 via the *narrow* method. Then assuming that we know the run (*run_id* variable) and event (*event_id* variable) numbers we can display the event. In order to perform this the *draw_event* method is called in line 26. If the run and event number are valid the event image can be displayed by calling *setIcon* method of the *javax.swing.JLabel* class as it is done in line 28. For simplicity the requests for the list of runs and list of events are not shown here, but they should be done in the same way as the *draw_event* request and valid run and event numbers shall be presented to a user in order to give him a possibility to chose which ones he is interesting in. For example the Visualizer application shown on the Figure 17 uses the *javax.swing.JTree* class to represent this information.

4.7 Running applications

The Event Painter can be started by issuing the following command in the Unix shell (assuming that the executable name is **event_painter**):

```
prompt> event_painter > Reference.file
```

This command starts the Painter application that prints the *Event_Painter_impl* objects reference to the 'Reference.file' file. This reference shall be used as a parameter for the Event Visualizer in order to let him access the Painter object.

```
prompt> java EventVisualizer `cat Reference.file`
```

The Visualizer application can be started on any machine - it is not necessary to run it on the same one on which the Painter is working. The only thing to worry about is the availability of the Reference.file file on that machine.

There is another way to establish connection between Event Visualizer and Event Painter without using intermediate file for the object reference storage. The CORBA Naming Service described in section 3.1.1 can be used to publish the Event Painter's object reference by associating it with some well known name. The Event Visualizer has to call the resolve method of Naming Service interface with this name as parameter in order to get the Painter reference.

The Figure 19 shows how Event Visualizer application looks like. It uses a tree to represent all the possible run numbers and events that belong to these runs. When the user selects an event number in the tree the *draw_event* method of the Event Painter is called and the image on the right side of the window is updated. The image shown by the Event Visualizer on the Figure 19 it created by the very simple Event Painter application. It does not paint a real event. It simply draws an image with the indication of the requested event and run numbers in order to illustrates the capacity for work of the proposed approach.

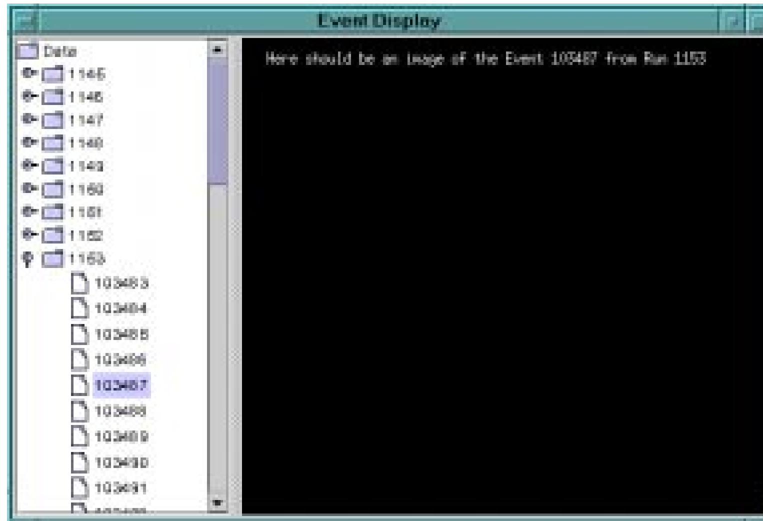


Fig. 19 Event Visualizer main window

4.8 Remarks about Java RMI

The Event Display example has been shown might be implemented with another distributed technology, for example with Java Remote Method Invocation (RMI)[33]. The Java RMI is an excellent communication technology in the Java language domain. Other languages can involved only via the Java Native Interface (JNI)[34] that is currently available for C and C++ only. For example the Event Painter task might be implemented in Java using JNI for the data storage access. But JNI is another technology to be learned and another layer to be added to an application increasing the overall system complexity. It worth thinking about using a CORBA broker with the natural C++ language mapping instead of calling the necessary C++ methods via the JNI.

What else should be taken into account while choosing the implementation technology is a legacy issue: it is not certain in 10 years we will still use Java. CORBA in contrary to Java is a standard that does not relay to a particular programming language and is able to assimilate a new languages as it has been done already with Java.

Nevertheless Java RMI fits perfectly to the modern object design patterns. It is more simple to learn and to use because the role of the Interface Definition Language is played by the Java language itself. The only serious drawback is the number of operating systems for which Java is available. But this limitation was partly resolved recently by the common efforts of Sun and OMG. Sun has implemented the RMI interface over the IIOP protocol[35] and OMG defines the mappings from Java language to the OMG IDL[36]. It is possible now to connect a client implemented with RMI and a server developed with CORBA and vice versa.

5. APPLICATIONS OF CORBA IN THE HEP ENVIRONMENT

Ordinarily, CORBA brokers are not used for the implementation of the on-line software that is responsible for the fast physical data transportation. The reason is the overhead of the ORB over pure network protocol communication (UDP or TCP/IP). This overhead is introduced by automatically generated stub and skeleton code that is executed for each remote method invocation and by another level of the communication protocol (generally IIOP) that is used by most of the ORBs for the interoperability reasons.

But for the control systems and off-line data access the CORBA implementations have been started to be used recently. Here there are a few references to the large HEP experiments in which CORBA brokers have been used.

5.1 ATLAS[37] Trigger/DAQ prototype -1 at CERN (Geneva)

The goal of the TDAQ Prototype -1 project[38] is to produce a prototype system representing a "full slice" of a DAQ suitable for evaluating candidate technologies and architectures for the final ATLAS DAQ system on the LHC accelerator at CERN. The back-end DAQ component of the project encompasses the software to configure, control and monitor the DAQ but excludes the processing and transportation of physics data. All the communications in back-end subsystem are implemented on top of Inter Language Unification (ILU) system. ILU is implemented by Xerox and can be thought of as a CORBA ORB system (though with omissions from and extensions to the CORBA specification). The detailed description of how CORBA is used for the ATLAS TDAQ prototype -1 can be found in [39].

5.2 Textor[40] plasma-physics experiment at Plasmaphysics Institute (Julich)

In this experiment CORBA has been used to implement an interface to the distributed database providing data access over Internet. This database contains the measurements data for the Textor-94 experiment and the current system is using the Objectivity database. More details can be found in [41].

5.3 PHENIX[42] on-line control system (Brookhaven National Laboratory)

The PHENIX detector at the Relativistic Heavy Ion Collider (RHIC) will study the dynamics of ultra-relativistic heavy ion collisions and search for exotic states of matter, most notably the Quark Gluon Plasma (QGP). The PHENIX online control system is responsible for the configuration, control and monitoring of the PHENIX detector data acquisition system and ancillary control hardware. The online system consists of a large number of embedded commercial and custom processors as well as custom software processes which are involved in the collection, monitoring and control of the detector and the event data. These processing elements are distributed over a diverse set of computing platforms including VME based Power PC controllers, Pentium based NT systems, and SUN Solaris SPARC processors. The IONA Technologies Orbix CORBA broker has been used as the communication mechanism for the PHENIX online system [43].

5.4 BaBar configuration databases (Stanford Linear Accelerator Center)

The BaBar[44] experiment at the Stanford Linear Accelerator Center is designed to study the CP violation in decays of B mesons produced in electron-positron interactions. BaBar has chosen an Object Oriented Database Management System, Objectivity/DB, as the underlying storage technology. The online system has also adopted Objectivity to store the ambient data and the configuring parameters of various hardware and software components of the detector. To provide access to the ambient data before and after they are stored in the database a CORBA interface has been developed and implemented[45]. It allows Java based browsers to analyze and display the data while they are being accumulated.

6. CONCLUSION

The OMG was founded in 1989 by 11 companies. Now it is composed of more then 900 members, among of which there are most of the leading software development companies. The first version of the CORBA standard (1.0) was issued in 1991. It included mostly the IDL definition and C language mapping. After that a new revision of the standard appeared almost each year. Based on the CORBA users' feedback all these revisions included important improvements like the interoperability architecture and Java mapping in the CORBA 2.0, POA specification in the CORBA 2.2, Java to IDL mapping in the CORBA 2.3. OMG has invested essential efforts to the integration with the other communication standards like COM/OLE and Java RMI.

All these efforts, have been invested by the OMG to the CORBA standard, result in an incredibly large number of ORB implementations. There are many good quality ORBs available now including free and commercial ones with a very wide range of operating systems and programming languages supported. The 10 years of CORBA evolution give an impressive example of a good quality standard development and maintenance. At the moment the CORBA standard is recognized as a very powerful

and useful object communication model by the programming community and it looks very likely that it will carry on this leading role in the software communication domain.

In the future plans of the OMG the most important issues are the Internet integration, the quality of service control support and CORBA component model development. More information about these categories can be found at the OMG announces Web page (http://sisyphus.omg.org/technology/corba/corba3releaseinfo.htm).

7. REFERENCES

[1] Unix Network Programming: Networking APIs: Sockets and Xti, W. Richard Stevens.

[2] Power Programming with RPC, John Bloomer, published by O'Reilly, 1992.

[3] Distributed Computing Environment homepage http://www.osf.org/dce/

[4] Open Software Foundation homepage http://www.osf.org/

[5] The Object Management Group official home page is http://www.omg.org/

[6] Object-oriented Analysis and Design with Applications, Grady Booch.

[7] Rose is a visual modeling tool of Rational Software, http://www.rational.com/products/rose/index.jtmpl.

[8] Software trough Pictures is a visual modeling framework of Aonix, http://www.aonix.com/content/products.html#stp

[9] Together is a Java, full UML modeler for Simultaneous Design-and-Code Editing of Togethersoft, http://www.togethersoft.com/together/matrix.html

[10] Advanced C++ Programming Styles and Idioms, James O. Coplien.

[11] CORBA/IIOP 2.3.1 Specification, chapter 3-IDL Syntax and Semantics, http://cgi.omg.org/cgi-bin/doc?formal/99-07-07

[12] CORBA Language Mapping Specifications Available Electronically, http://www.omg.org/technology/documents/formal/corba_language_mapping_specifica.htm

[13] CORBA/IIOP 2.3.1 Specification, http://www.omg.org/technology/documents/formal/corba_2.htm

[14] OMG IDL to Java Language Mapping, Formal/99-07-53, http://www.omg.org/technology/documents/formal/omg_idl_to_java_language_mapping.htm

[15] The Document company XEROX, http://www.parc.xerox.com/parc-go.html

[16] Inter-Language Unification, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

[17] CORBA/IIOP 2.3.1 Specification, chapter 7-Dynamic Invocation Interface, http://cgi.omg.org/cgi-bin/doc?formal/99-07-11

[18] CORBA/IIOP 2.3.1 Specification, chapter 8-Dynamic Skeleton Interface, http://cgi.omg.org/cgi-bin/doc?formal/99-07-12

[19] CORBA/IIOP 2.3.1 Specification, chapter 11-Portable Object Adapter, http://cgi.omg.org/cgi-bin/doc?formal/99-07-15

[20] CORBA/IIOP 2.3.1 Specification, chapter 4-ORB Interface, http://cgi.omg.org/cgi-bin/doc?formal/99-07-08

[21] CORBA/IIOP 2.3.1 Specification, chapter 12 - Interoperability Overview, http://cgi.omg.org/cgi-bin/doc?formal/99-07-16

[22] CORBA/IIOP 2.3.1 Specification, chapter 15 General Inter-ORB Protocol, http://cgi.omg.org/cgi-bin/doc?formal/99-10-11

[23] The complete Discussion of the OMA, formal/00-06-41, http://www.omg.org/technology/documents/formal/object_management_architecture.htm

[24] CORBA Common Facilities Specifications, http://www.omg.org/technology/documents/formal/corba_common_facilities_specific.htm

[25] CORBA Services, OMG formal documents, http://www.omg.org/technology/documents/formal/corba_services_available_electro.htm

[26] Naming Service, version 1.0, http://www.omg.org/technology/documents/formal/naming_service.htm

[27] The Java IDL tutorial, http://java.sun.com/docs/books/tutorial/idl/index.html
[28] Java 2 SDK, Standard Edition Documentation, http://www.javasoft.com/products/jdk/1.2/docs/index.html
[29] ORBacus for C++ and Java, http://www.ooc.com/products/orbacus.html
[30] Object Oriented Concepts, Inc., Canada, http://www.ooc.com/
[31] A freely distributed tool for converting IDL interface definitions to Java stub and skeleton files, available at http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html
[32] Extensible Markup Language (XML) 1.0, http://www.w3.org/TR/1998/REC-xml-19980210
[33] Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi/index.html
[34] Java Native Interface, http://java.sun.com/j2se/1.3/docs/guide/jni/index.html
[35] RMI over IIOP 1.0.1, http://www.javasoft.com/products/rmi-iiop/index.html
[36] Java Language Mapping to OMG IDL, http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
[37] ATLAS homepage http://atlasinfo.cern.ch:80/Atlas/Welcome.html
[38] ATLAS Trigger/DAQ Prototype-1 homepage http://atddoc.cern.ch/Atlas/
[39] Applications of Corba in the Atlas prototype DAQ, S. Kolos, R. Jones. L.Mapelli, Y. Ryabov, 11th IEEE NPSS Real Time Conference Proceedings, 1999, pp 469-474
[40] Textor-94 experiment homepage is http://www.fz-juelich.de/ipp
[41] Objectivity / Corba distributed database performance on gigabit SUN-Ultra-10 cluster, L.Gommans and others, 11th IEEE NPSS Real Time Conference Proceedings, 1999, 442-445
[42] Overview of PHENIX Online System, C.Witzig, 10th IEEE Real Time Conference Proceedings, 1998, pp 541-543
[43] Use of CORBA in the PHENIX Distributed Online Computing System, E.Desmond and others, 11th IEEE NPSS Real Time Conference Proceedings, 1999, pp 487-491
[44] BaBar homepage http://www.slac.stanford.edu/BFROOT/
[45] Ambient and Configuration Databases for the BaBar Online System, G. Zioulas and others, 11th IEEE NPSS Real Time Conference Proceedings, 1999, pp 548-550

A CACHE-BASED DATA INTENSIVE DISTRIBUTED COMPUTING ARCHITECTURE FOR "GRID" APPLICATIONS

Brian Tierney, William Johnston, Jason Lee
Lawrence Berkeley National Laboratory, Berkeley, CA 94720

Abstract
Modern scientific computing involves organizing, moving, visualizing, and analyzing massive amounts of data from around the world, as well as employing large-scale computation. The distributed systems that solve large-scale problems will always involve aggregating and scheduling many resources. Data must be located and staged, cache and network capacity must be available at the same time as computing capacity, etc. Every aspect of such a system is dynamic: locating and scheduling resources, adapting running application systems to availability and congestion in the middleware and infrastructure, responding to human interaction, etc. The technologies, the middleware services, and the architectures that are used to build useful high-speed, wide area distributed systems, constitute the field of data intensive computing, and are sometimes referred to as the "Data Grid". This paper explores the use of a network data cache in a Data Grid environment.

1. INTRODUCTION

High-speed data streams resulting from the operation of on-line instruments and imaging systems are a staple of modern scientific, health care, and intelligence environments. The advent of high-speed networks is providing the potential for new approaches to the collection, organization, storage, analysis, visualization, and distribution of the large-data-objects that result from such data streams. The result will be to make both the data and its analysis much more readily available.

For example, high energy physics experiments generate high rates and massive volumes of data that must be processed and archived in real time. This data must also be accessible to large scientific collaborations — typically hundreds of investigators at dozens of institutions around the world.

In this paper we will describe how "Computational Grid" environments can be used to help with these types of applications, and give a specific example of a high energy physics applications in this environment. We describe how a high-speed application-level network data cache is a particularly important component in a data intensive grid architecture, and describe our implementation of such a cache.

2. DATA INTENSIVE GRIDS

The integration of the various technological approaches being used to address the problem of integrated use of dispersed resources is frequently called a "grid," or a computational grid — a name arising by analogy with the grid that supplies ubiquitous access to electric power. See, e.g., [9]. Basic grid services are those that locate, allocate, coordinate, utilize, and provide for human interaction with the various resources that actually perform useful functions.

Grids are built from collections of primarily independent services. The essential aspect of grid services is that they are uniformly available throughout the distributed environment of the grid. Services may be grouped into integrated sets of services, sometimes called "middleware." Current grid tools include Globus [8], Legion [15], SRB [2], and workbench systems like Habanero [10] and WebFlow [1]. Recently the term "Data Grid" has come into use to describe middleware and services for data intensive Grid applications [3], and several data grid research projects have be started [5][[17].

From the application's point of view, the Grid is a collection of middleware services that provide applications with a uniform view of distributed resource components and the mechanisms for assembling them into systems. From the middleware systems points of view, the Grid is a standardized set of basic services providing scheduling, resource discovery, global data directories, security, communication services, etc. However, from the Grid implementor's point of view, these services result from and must interact with a heterogeneous set of capabilities, and frequently involve "drilling" down through the various layers of the computing and communications infrastructure.

2.1 Architecture for Data Intensive Environments

Our model is to use a high-speed data storage cache as a common element for all of the sources and sinks of data involved in high-performance data systems. We use the term "cache" to mean storage that is faster than typical local disk, and temporary in nature. This cache-based approach provides standard interfaces to a large, application-oriented, distributed, on-line, transient storage system. In a wide-area Grid environment, the caches must be specifically designed to achieve maximum throughput over high-speed networks.

Each data source deposits its data in the cache, and each data consumer takes data from the cache, often writing the processed data back to the cache. A tertiary storage system manager migrates data to and from the cache at various stages of processing. (See Figure1.) We have used this model for data handling systems for high energy physics data and for medical imaging data. These applications are described in some detail in [14] and [13].

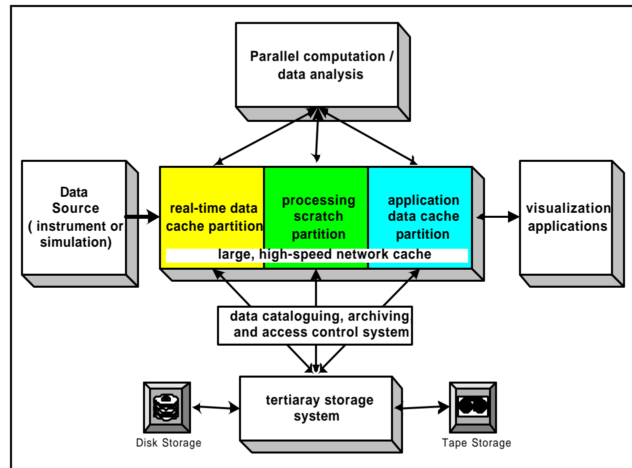


Figure 1 The Data Handling Model

The high-speed application-level cache serves several roles in this environment. It provides a standard high data rate interface for high-speed access by data sources, processing resources, mass storage systems (MSS), and user interface / data visualization elements. It provides the functionality of a single very large, random access, block-oriented I/O device (i.e., a "virtual disk"). This cache also serves to isolate the application from tertiary storage systems and instrument data sources, helping eliminate contention for those resources.

This cache can be used as a large buffer, able to absorb data from a high rate data source and then to forward it to a slower tertiary storage system. The cache also provides an "impedance matching" function between a small number of high throughput streams to a larger number of lower speed streams, e.g. between fine-grained accesses by many applications and the coarse-grained nature of a few parallel tape drives in the tertiary storage system.

Depending on the size of the cache relative to the objects of interest, the tertiary storage system management may only involve moving partial objects to the cache. In other words, the cache may contain a moving window for an extremely large off-line object/data set. Generally, the cache storage configuration is large (e.g., 100s of gigabytes) compared to the available disks of a typical computing

environment (e.g., 10s of gigabytes), and very large compared to any single disk (e.g. hundreds of ~10 gigabytes).

In this type of environment, a client typically will copy large portions of a data set from the remote archive to local disk before visualizing or processing the data. However, if the network is fast enough and if the cache is tuned for remote access and uses parallel disks, the cache can provide data access to remote clients that is even faster than local disk. Additionally many applications actually only need a small portion of the total data set. By leaving the data on a remote cache, a much smaller amount of data may actually be moved over the network.

3. THE DISTRIBUTED-PARALLEL STORAGE SYSTEM

Our implementation of this high-speed, distributed cache is called the Distributed-Parallel Storage System (DPSS) [20]. LBNL designed and implemented the DPSS as part of the DARPA MAGIC project [6], and as part of the U.S. Department of Energy’s high-speed distributed computing program. This technology has been successful in providing an economical, high-performance, widely distributed, and highly scalable architecture for caching large amounts of data that can potentially be used by many different users.

Typical DPSS implementations consist of several low-cost workstations as DPSS block servers, each with several disk controllers, and several disks on each controller. A four-server DPSS with a capacity of one Terabyte (costing about $10-$12K in mid-2000) can thus produce throughputs of over 70 MBytes/sec by providing parallel access to 20-30 disks.The overall architecture of the DPSS is illustrated in Figure2.

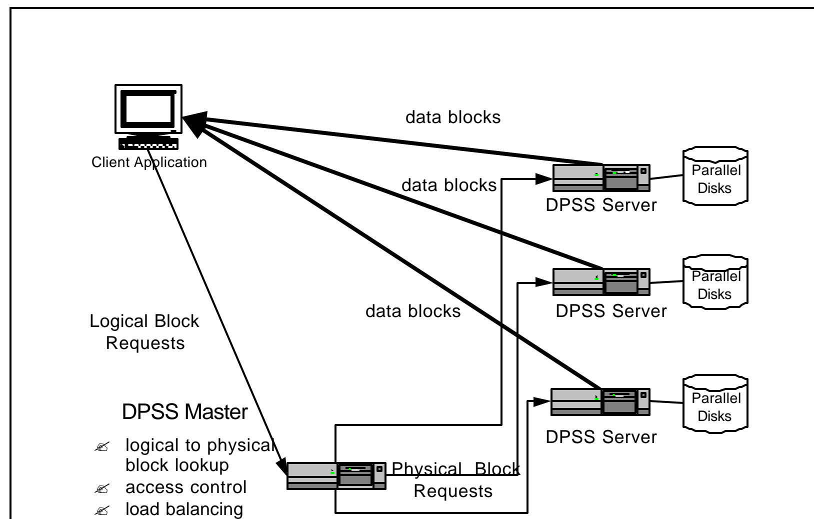


Figure 2 Overall DPSS Architecture

Other papers describing the DPSS in more detail include [20], which describes how the DPSS was used to provide high-speed access to remote data for a terrain visualization application, [21], which describes the basic architecture and implementation, and [22], which describes how the instrumentation abilities in the DPSS were used to help track down a wide area network problem.

The application interface to the DPSS cache supports a variety of I/O semantics, including Unix-like I/O semantics, through an easy to use client API library (e.g. dpssOpen(), dpssRead(), dpssWrite(), dpssLSeek(), dpssClose()). The data layout on the disks is completely up to the application, and the usual strategy for sequential reading applications is to write the data “round-robin,” striping blocks of data across the servers. The client library also includes a flexible data replication ability, allowing for multiple levels of fault tolerance. The DPSS client library is multi-threaded, where the number of client threads is equal to the number of DPSS servers. Therefore the speed of the client scales with the speed of the server, assuming the client host is powerful enough.

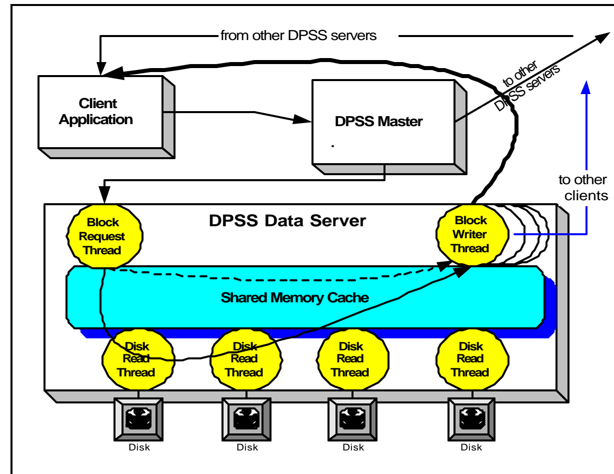


Figure 3 DPSS Server Architecture

The internal architecture of the DPSS is illustrated in Figure3. Requests for blocks of data are sent from the client to the "DPSS master" process, which determines which "DPSS block servers" the blocks are located on, and forwards the requests to the appropriate servers. The server then sends the block directly back to the client. Servers may be anywhere in the network: there is no assumption that they are all at the same location, or even the same city.

DPSS performance, as measured by total throughput, is optimized for a relatively smaller number (a few thousand) of relatively large files (greater than 50 MB). Performance is the same for any file sizes greater than 50 MB. We have also shown that performance scales well with the number of clients, up to at least 64 clients. For example, if the DPSS system is configured to provide 50 MB/sec to 1 client, it can provide 1 MB/sec to each of 50 simultaneous clients. The DPSS master host starts to run out of resources with more than 64 clients.

Because of the threaded nature of the DPSS server, a server scales linearly with the number of disks, up to the network limit of the host (possibly limited by the network card or the CPU). The total DPSS system throughput scales linearly with the number of servers, up to at least 10 servers.

The DPSS provides several important and unique capabilities for data intensive distributed computing environments. It provides application-specific interfaces to an extremely large space of logical blocks; it offers the ability to build large, high-performance storage systems from inexpensive commodity components; and it offers the ability to increase performance by increasing the number of parallel disk servers.

DPSS data blocks are available to clients immediately as they are placed into the cache. It is not necessary to wait until the entire file has been transferred before requesting data. This is particularly useful to clients requesting data from a tape archive. As the file moves from tape to the DPSS cache, the blocks in the cache are immediately available to the client. If a block is not available, the application can either block, waiting for the data to arrive, or continue to request other blocks of data which may be ready to read.

3.1 TCP Issues

The DPSS uses the TCP protocol for data transfers. For TCP to perform well over high-speeds networks, it is critical that there be enough buffer space for the congestion control algorithms to work correctly [11]. Proper buffer size is a function of the network bandwidth-delay product, but because bandwidth-delay products in the Internet can span 4-5 orders of magnitude, it is impossible to configure the default TCP parameters on a host to be optimal for all connections [18].

Figure4 shows the importance of the setting of the TCP buffer size correctly. This figure illustrates that buffers can be hand-tuned for either LAN access or WAN access, but not both at once. It is also apparent that while setting the buffer size big enough is particularly important for the WAN case, it is also important not to set it too big for the LAN environment. If the buffers are too large,

throughput may decrease because the larger receive buffer allows the congestion window to grow sufficiently large that multiple packets are lost (in a major buffer overflow) during a single round trip time (RTT), which then leads to a time-out instead of a smooth fast retransmit/recovery.

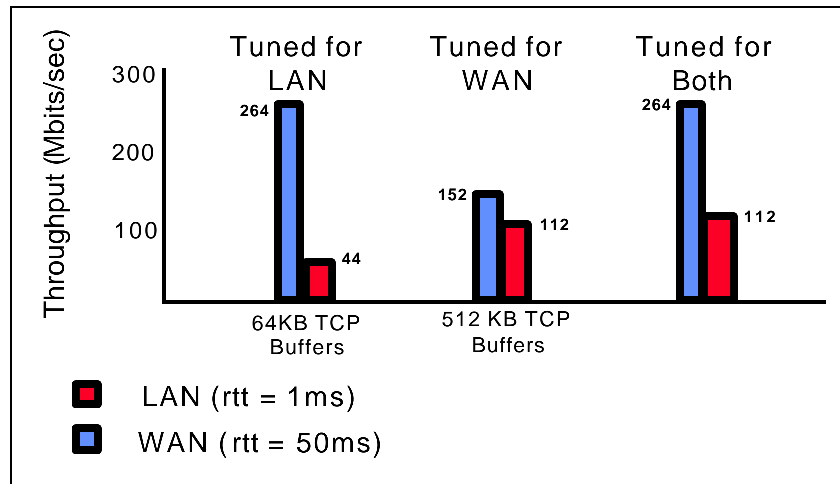


Figure 4 Importance of correct TCP tuning

To solve this problem, the DPSS client library attempts to automatically determine the bandwidth-delay product for each connection to a DPSS server and sets the TCP buffer size to the optimal value dynamically for each client. This provides optimal tuning both LAN and WAN clients simultaneously. Optionally the user can specify the buffer size to the DPSS client library via a shell environment variable.

We are currently developing a network monitoring service that will monitor the network path between specified sites, and store the bandwidth delay products for these network paths in a directory service. Then "network-aware" application can query this service for the optimal TCP buffer size to use on the fly.

4. A HIGH ENERGY PHYSICS APPLICATION

We have conducted a set of high-speed, network based, data intensive computing experiments between Lawrence Berkeley National Laboratory (LBNL) in Berkeley, Calif., and the Stanford Linear Accelerator (SLAC) in Palo Alto, Calif. The results of this experiment were that a sustained 57 megabytes/sec of data were delivered from datasets in the distributed cache to the remote application memory, ready for analysis algorithms to commence operation. This experiment represents an example of our data intensive computing model in operation.

The prototype application was the STAR analysis system that analyzes data from high energy physics experiments. (See [7].) A four-server DPSS located at LBNL was used as a prototype front end for a high-speed mass storage system. A 4-CPU Sun E-4000 located at SLAC was a prototype for a physics data analysis computing cluster, as shown in Figure5. The National Transparent Optical Network testbed (NTON - see [16]) connects LBNL and SLAC and provided a five-switch, 100-km, OC-12 ATM path. All experiments were application-to-application, using TCP transport.

Multiple instances of the STAR analysis code read data from the DPSS at LBNL and moved that data into the memory of the STAF application where it was available to the analysis algorithms. This experiment resulted in a sustained data transfer rate of 57 MBytes/sec from DPSS cache to application memory. This is the equivalent of about 4.5 TeraBytes / day. The goal of the experiment was to demonstrate that high-speed mass storage systems could use distributed application-level caches to make data available to the systems running the analysis codes. The experiment was successful, and the next steps will involve completing the mechanisms for optimizing the MSS staging patterns and completing the DPSS interface to the bit file movers that interface to the MSS tape drives.

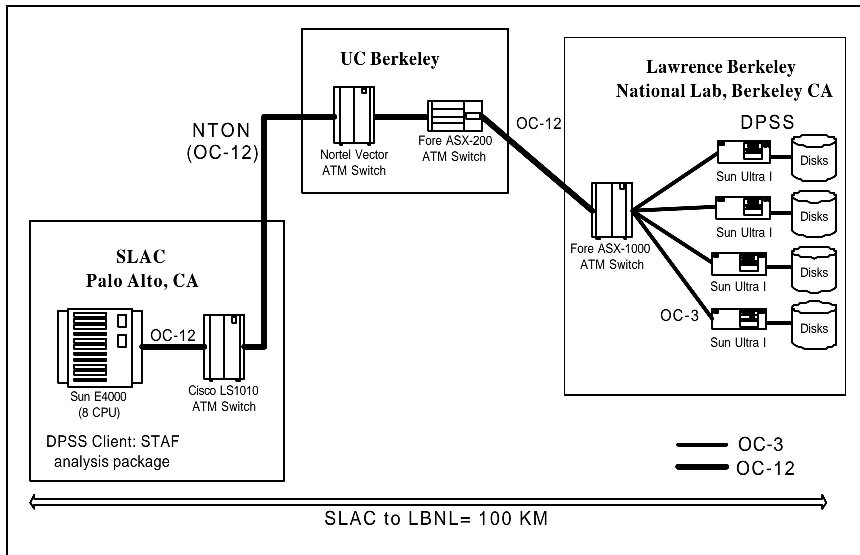


Figure 5 HEP Experiment Configuration

4.1 Current Results

We recently achieved 71 MBytes/sec (568 Mbits/sec) of I/O across the wide area using the configuration shown in Figure6. This was using a single application runing on an 8 node Compac Alpha Linux cluster, located at Sandia National Lab in Livermore, CA, about 100 km from the DPSS system at Lawrence Berkeley National Lab in Berkeley, CA. Note that the bottleneck link in the network is an OC-12 "packet over Sonet" (i.e.: not ATM) pipe from Berkeley to Oakland. The maximum data rate on OC-12 ATM is 534 Mbits/sec due to ATM header overhead, which OC-12 packet over Sonet does not have. This is how we were able to acheive a throughput that is faster that the maximum OC-12 ATM data rate.

It should be noted that both this application and the HEP application above read data from the DPSS in chunk of at least 8 MBytes per dpssRead() call. Read block sizes on the order of at least 4 MBytes in size are required achieve such high I/O rates, in order to fill the entire I/O pipeline.

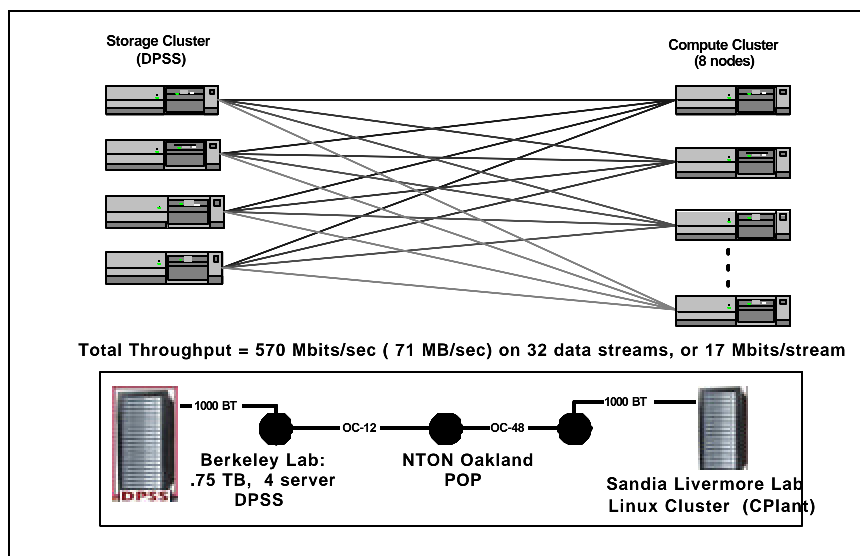


Figure 6 Compute Cluster Application

5. CONCLUSIONS

We believe this architecture, and its integration with systems like Globus, will enable the next generation of configurable, distributed, high-performance, data-intensive systems; computational steering; and integrated instrument and computational simulation. We also believe a high performance network application-level cache system such as the DPSS will be an important component to these "computational grid" and "data grid" environments.

Acknowledgments

The work described in this paper is supported by the U. S. Dept. of Energy, Office of Energy Research, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences and ERLTT Divisions under contract DE-AC03-76SF00098 with the University of California, and by DARPA, Information Technology Office.

6. REFERENCES

[1] Erol Akarsu, Geoffrey C. Fox, Wojtek Furmanski, Tomasz Haupt, "WebFlow - High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing," Proceedings of IEEE Supercomputing '98, Nov. 1998.

[2] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, Michael Wan, "The SDSC Storage Resource Broker," Proc. CASCON'98 Conference, Nov.30-Dec.3, 1998, Toronto, Canada. (http://www.npaci.edu/DICE/SRB)

[3] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets", Internet II Network Storage Symposium, Oct. 1999, http://dsi.internet2.edu/netstore99/

[4] K. Czajkowski, I. Foster, C., Kesselman, N. Karonis, S. Martin, W. Smith, S. Tuecke. "A Resource Management Architecture for Metacomputing Systems," Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,1998.

[5] EU DATAGRID Project, http://home.cern.ch/~les/grid/

[6] B. Fuller and I. Richer "The MAGIC Project: From Vision to Reality," IEEE Network, May, 1996, Vol. 10, no. 3. http://www.magic.net/

[7] W. Greiman, W. E. Johnston, C. McParland, D. Olson, B. Tierney, C. Tull, "High-Speed Distributed Data Handling for HENP," Computing in High Energy Physics, April, 1997. Berlin, Germany. http://www-itg.lbl.gov/STAR/

[8] Globus: See http://www.globus.org

[9] Grid: *The Grid: Blueprint for a New Computing Infrastructure*, edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pub. August 1998. ISBN 1-55860-475-8. http://www.mkp.com/books_catalog/1-55860-475-8.asp

[10] Habanero: http://www.ncsa.uiuc.edu/SDG/Software/ Habanero/

[11] V. Jacobson, "Congestion Avoidance and Control," Proceedings of ACM SIGCOMM '88, August 1988.

[12] W. Johnston, G. Jin, C. Larsen, J. Lee, G. Hoo, M. Thompson, B. Tierney, J. Terdiman, "Real-Time Generation and Cataloguing of Large Data-Objects in Widely Distributed Environments," International Journal of Digital Libraries - Special Issue on "Digital Libraries in Medicine". November, 1997. (Available at http://www-itg.lbl.gov/WALDO/)

[13] William E. Johnston. "Real-Time Widely Distributed Instrumentation Systems," In *The Grid: Blueprint for a New Computing Infrastructure*. Edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pubs. August 1998.

[14] William E. Johnston, W. Greiman, G. Hoo, J. Lee, B. Tierney, C. Tull, and D. Olson. "High-Speed Distributed Data Handling for On-Line Instrumentation Systems," Proceedings of ACM/IEEE SC97: High Performance Networking and Computing. Nov., 1997. http://www-itg.lbl.gov/ ~johnston/papers.html

[15] Legion: See http://www.cs.virginia.edu/~legion/

[16] NTON, "National Transparent Optical Network Consortium." See http://www.ntonc.org/.

[17] Partical Phyics Data Grid (PPDG) Project, http://www.cacr.caltech.edu/ppdg/

[18] J. Semke, J. Mahdavi, M. Mathis, “Automatic TCP Buffer Tuning,” Computer Communication Review, ACM SIGCOMM, volume 28, number 4, Oct. 1998.

[19] M. Thompson, W. Johnston, J. Guojun, J. Lee, B. Tierney, and J. F. Terdiman, “Distributed health care imaging information systems,” PACS Design and Evaluation: Engineering and Clinical Issues, SPIE Medical Imaging 1997. (Available at http://www-itg.lbl.gov/Kaiser.IMG)

[20] Brian Tierney, William E. Johnston, Hanan Herzog, Gary Hoo, Guojun Jin, Jason Lee, Ling Tony Chen, Doron Rotem. “Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers,” ACM Multimedia ‘94 (San Francisco, October 1994). http://www-itg.lbl.gov/DPSS/papers/

[21] Brian Tierney, W. Johnston, H. Herzog, G. Hoo, G Jin, and J. Lee, “System Issues in Implementing High Speed Distributed Parallel Storage Systems,” Proceedings of the USENIX Symposium on High Speed Networking, Aug. 1994, LBL-35775. http://www-itg.lbl.gov/DPSS/ papers.html.

[22] Brian Tierney, W. Johnston, G. Hoo, J. Lee, “Performance Analysis in High-Speed Wide-Area ATM Networks: Top-to-Bottom End-to-End Monitoring,” IEEE Network, May, 1996, Vol. 10, no. 3. LBL Report 38246, 1996. http://www-itg.lbl.gov/DPSS/papers.html

IMPROVING QUALITY OF SERVICE IN THE INTERNET

François Fluckiger
CERN, Geneva, Switzerland

Abstract
The Internet transport technology was designed to be robust, resilient to link or node outages, and with no single point of failure. The resulting connectionless system supports what is called a "best effort datagram delivery service", the performance of which is often greatly unpredictable. To improve the predictability of IP-based networks, several Quality of Service technologies have been designed over the past decade. The first one, RSVP, based on reservation of resources, is operational but has several major deficiencies, such as scalability difficulties. However, associated to other more recent technologies -RSVP aggregation, Diffserv and MPLS- the combination may result into an appropriate solution for improving Quality of Service guarantees in a scalable way.

1. BACK TO BASICS: THE INTERNET TRANSPORT SERVICE

One of the chief reasons for the success of the Internet lies in its *transport* technology. What is so special with the Internet transport technology? Unlike all the other wide area network technologies which predated it –such as the Switched Telephone Network, X.25 ATM or Frame Relay -, the Internet is not connection-oriented. This means that no prior connection between the two communicating systems –we say the "*end-systems*"- is required before a first fragment of information –the *packet*, also called the *datagram*- can be sent. As a result, all packets, which are treated independently by the network, must carry the full address of the destination system.

Packets may also take differing *routes* to reach the destination, though in practice, this only happens in case of link or node failure. If they take differing routes, they may be delivered in a miss-ordered way, as a given packet may take over another one on the different path. More importantly, packets may be lost, for reasons we examine later. If a packet is lost, the end-systems are not informed, and they have to detect such loses by themselves –as long as it matters for them to know about data losses. What the network guarantees is that it will do "*its best*" to deliver these datagrams. This is why the resulting Internet transport service is called a *best-effort datagram delivery service*.

Each of these base techniques, *connection-less* and *connection-oriented* have their own merits and drawbacks.

- In a *connection-less* network, there is no delay needed to set up a connection between the two communicating systems, before a first data packet can be sent. This is by the way, a fundamental feature of the Internet transport technology which was key in the support of the main Internet application: The World Wide Web. Indeed, imagine if, when you click on a link, a hard connection –that is a sort of telephone call- had be set over the network before the first packet of your request be sent to the destination (well, in practice, there is a first handshake called the TCP connection, but this is very fast as it only involves the two end-systems, but not the underlying network). This would be by far too slow. There would be no web!
- Another major advantage is that the routing is more dynamic as packets are independent,

and the network may easily adapt to changing conditions without impacting the service, as packet reroutes may remain unnoticed. Thus, the network appears as more resilient to failures.

Conversely, connection-oriented networks have also advantages.

- Connection-oriented networks, being aware of the requests before conversations actually start, by means of the *connection set up* procedure, may better predict the traffic load.
- If the traffic is more predictable, *resources* may be *reserved* more easily.
- If not enough resources are available, then the connection is refused. This is called the "busy signal", the functionality by which the network tells the user "*Sorry, I can't accept you call for the moment, please try later*". This is what we experience with the telephone network, the oldest connection-oriented service. This process is called "*Call Admission Control*" (*CAC*) or also "*Capacity Admission Control*" because what matters for deciding whether a new request for connection is granted is the available capacity of the network.
- It is therefore easier for connection-oriented networks to guarantee the quality of service of the communications they support.

2. WHY IMPROVING QUALITY OF SERVICE?

The objective of the efforts undertaken since the beginning of the 90s about the Internet quality of service is to improve the predictability of the service. Indeed, the historical "best effort datagram service" results in a somewhat unpredictable behaviour. There are multiple reasons why this has become no longer desirable.

- Users may wish to set up *Virtual Private Networks* (*VPN*) over the shared Internet with a guaranteed Quality of Service, such as the bandwidth of the pipes between sites part of the VPN. For example, imagine a company with one head-quarter and three branches, all four connected to the same *Internet Service Provider* (*ISP*) - to simplify the case. The VPN is to be made of three links between the branches and the head-quarter, each with a guaranteed bandwidth of say, 1 Megabit/second. Can we do that with the current Internet, that is with regular routers? No, we can't. We need something more than the "best effort datagram service"
- Users which connect to an ISP at a given access speed may wish to have a secure aggregate bandwidth out of this access link, irrespective of the destination of their traffic. For example, a company connecting to an ISP at 1.5 Megabit/second, and thus paying for that access speed, may wish to be guaranteed at least 2/3 of this access bandwidth for all its outgoing traffic, wherever it goes. Again, we can't do this with the "best effort datagram service".
- More and more multimedia applications use the Internet, in particular, audio and video streams. These streams usually need a minimum bit rate, below which it makes no sense to try and send the audio or video traffic. These are requirements that do not apply to aggregates of traffic as in the above case, but to point-to-point flows between two end-systems.

3. SERVICE DISCRIMINATION

Thus, the efforts for improving the *Quality of Service* (*QoS*) guarantees aim at moving away from the historical model of traffic where all packets are handled with the same priority by the network. By abandoning the pure egalitarian treatment of the datagrams, the new Quality of Service techniques create discrimination between packets. This is called *service discrimination*.

Service discrimination does not create any resource by itself –we do not get more bit rate on a link because some packets have higher priorities- therefore, it is not solving all problems of Quality of

Service. If a network, or a portion of a network (a link), has not enough capacity, service discrimination will not help for all the traffic. However it will help for some. Indeed, the objective of service discrimination is to give better service to some traffic. But this is done at the expense of giving a worse service to the rest. Hopefully, this only occurs in *times of congestion*.

In passing, this move from an egalitarian world, where all packets were equal, to a world where some packets are "more equal than others", a world where discrimination is legalized or organized, was viewed by some as orthogonal to the evolution of our society. I will leave this discussion to the judgement of the reader …

4. INTEGRATED SERVICES

The first substantial work on Quality of Service in the Internet started in the early 90s in the framework of what was called the *Integrated Services* (*IS*) model. The first release was made in 93.

The Integrated Services model is based on the statement that a single *class* of packets is no longer sufficient, and that new classes with higher priorities are needed, in the same way as we have the economy, business and often first class with airlines. How many new classes were needed? The Integrated Services model opted for two new classes of packets, resulting in a total of three possible classes in the new discriminated Internet world:

- The *best effort service class* (*BE*)

  This is the default class

- The *controlled-load service class* (*CS*)

  There, if the sender respects a certain traffic profile (that is a certain bit rate) for a given flow, then the network promises to behave as though it was *unloaded*, but without *quantitative* guarantees in particular of the latencies of the packets.

- The *guaranteed service class* (*GS*)

  There, packets are promised to be delivered within a *firmly bounded delay*. This is for special applications with very stringent time delivery requirements.

5. RESOURCE RESERVATIONS

The guiding principles of the Integrated Services model are the following:

- **Resource reservation is necessary**

  To improve the guarantees, the key resources needed in the network must be *reserved* in some way.

- **Reservations operate on flows**

  A *flow* is a stream of packets between one source and one destination (note that the destination may be a unique destination in the usual cases, but also a multiple destination, in case of a multicast flow; but this is comment for the specialists). For every flow that needs to benefit from either the *CS* or the *GS* service, reservations need be made.

- **Routers have to maintain flow-specific states**

  By *state*, we mean a block of memory in the router where information about the flow and its requirements are stored: the service class (*CS* or *GS*), the bit rate to guarantee for that flow, the conditions for delay if applicable, …

- **Dynamic Reservations need a signaling (set-up) protocol:**

This protocol has been specified and is called *Resource Reservation Protocol*, or *RSVP*

6. HOW TO EXPRESS QUALITY OF SERVICE

For a given flow, the *parameters* for determining the Quality of Service belong to two groups. The two groups reflect a *contract* between the user of the network on the one hand -or more exactly the user *sending* the traffic- and the network on the other hand.

- **Traffic parameters**

  The first group specifies what the flow *traffic pattern* is to be: parameters include the *sustained bit rate*, and the *burst size*, that is the tolerance that exists for the sender to exceed for short periods of time the sustained bit rate -as long as over longer periods, the average bit rate remains within the limit of the sustained bit rate.

  The sender promises in the contract to respect this traffic profile.

- **Quality parameters**

  These parameters specify the quality the network promises in turn to guarantee if the sender respects its agreed traffic profile. There are two things the network can possibly promise:

  - *latencies*, that is guarantees on the delay it takes for packets to traverse the network; this is called the *transit delay*
  - *loss ratio*: that the maximum proportion of packets not delivered against the total number of packets sent.

7. RSVP GUIDING PRINCIPLES

The *Resource Reservation Protocol, RSVP,* is the mechanism defined by the Integrated Services for reserving resources in the network. It is called a *signaling* protocol, because its aim is to *signal* to the network that a given flow is going to require certain guarantees for latencies and loss ratio, if the flow respects a certain bit rate. RSVP is based on a number of guiding principles.

- **RSVP is to co-exist with regular datagram service**

  Any router which supports RSVP, also supports the regular best effort datagram service

- **RSVP does not set hard connections**

  Instead, the connections are said to be "soft", and we explain this concept a bit later.

- **The amount of reserved resources is recipient driven**

  That is, the sender will only *propose* a certain traffic profile. But this is the receiver system which will *decide* how much of this proposed bit rate it can accommodate. Indeed, a frequent case may be when the sender is a fast powerful server system and the receiver a slow desktop device. The capabilities of the receiver to process the received data, for example to decompress a video stream, may be much lower than that of the sending server.

- **Reservations are unidirectional**

  If an application requires bi-directional reservations (such as in telephony, which is of course different from viewing a movie over the Internet, which is one-way only application), two reservations will have to be made, one in every direction.

8. RSVP PROTOCOL MECHANISMS

The RSVP protocol is only concerned with conveying information -along a path followed by a flow- to routers so that they can reserve the resources they need. For this, the protocol uses special RSVP *control packets*, which are packets which will be recognized by *RSVP-aware routers* and treated as such. The two main RSVP control packets are the *PATH* message and the *RESV* (reservation) message. The mechanisms for reserving resources for a given flow between a sender and a receiver are as follows:

- The “PATH” control messages are to be sent periodically by the sender

  This message carries the traffic parameters and the details of the requested service class. It has to be repeated periodically (thus, the sender keeps "saying" to the network "*I still need this quality of service for that particular flow*").
- The “PATH” control messages establishes an RSVP state in the intermediary routers
- The receiver replies with a “RESV” message, according to its capabilities
- The “RESV” message reserves resources, if available, in routers on the route back.
- I not enough resource are available at a given router, the router generates an error control message to the end-systems.
- If “PATH” is not repeated after time-out, then the resources are released
- “PATH” and “RESV” messages are carried by ordinary best-effort datagrams

9. WHICH RESOURCES ARE RESERVED

One aspect of RSVP which may be surprising at first sight is that the protocol which aims at reserving resource does not define what those resources are. And there is a good reason for this. The "resources”, that is "what is important" for a router or for a given transmission medium between two routers to guarantee a certain performance is fully implementation dependent. In particular, RSVP makes no assumption on the internals of routers.

That being said, in practice, with today’s routers and transmission links, reservations generally apply to two types of resources:

- a slice of the link bandwidth
- a fraction of the buffers from the buffer pool of the routers

Note that *reservation* is different from *allocation*. Reserving means that a given amount has been secured for a flow, but the exact and precise resource is not allocated (yet). This is similar to reservation in public transportation networks: reserving a flight ticket is different from getting the seat allocated.

10. PATH STABILITY

A difficult problem with RSVP lies with the fact that resources are reserved for a given flow over a given path. This path is the one followed by the initial PATH and RESV messages. However, it may turn out that the route followed by the packets change after the reservation has been made. This happens in particular if the route used for the first reservation was a long path, because a shorter path between the two end-systems was unavailable at that time (due to a router or link failure). When the

shortest path is back to availability, the data will naturally take this shortest route because this is how the regular Internet routing works: packets always try and take the shortest route.

We then have the reservations made on the longer initial route and the data flowing on a new shortest route where no reservation has been made. Fortunately, the situation may clear itself after a while, because PATH messages are repeated (roughly every minute). The next repeated message will then take the shortest route and reserve resources on this new route.

This is unfortunately not a complete solution, because it may turn out that there are not enough resources available on the shortest route. Then, the two users experience a situation where, without knowing why, they move from a good quality transmission to a bad one, simply because rerouting took place. What we need for a more complete solution is a means of having more stability for reserved paths, but also for finding routes, possibly longer, but which do have the required resources when the shortest route can not satisfy a given request. Such a routing technique for not only finding a short route but also one with enough resources to satisfy a certain quality of service is called *Quality of Service-based routing*.

11. RSVP OVERHEAD FOR DATA PACKETS

Another difficulty with RSVP lies in the *overhead* created in every router by the need to analyze every packet in order to know which priority it has, or more exactly which *service class* it belongs to (Best effort, Controlled-load, Guaranteed).

The process of deciding which service class an incoming packet belongs to is called *classification*. In the case of RSVP, how can we know the class? Is there any mark somewhere in the packet header, any field than can be rapidly and easily examined and that would tell us*: "It is class 1, 2 or 3*"? The answer is no: the class is not explicitly stated in the data packets when using RSVP only.

This is by examining the pair of source and destination addresses (and possibly other fields such a the protocol type) that characterizes every flow and by looking at a table to check whether a reservation has been made for that particular pair, that the router knows the class of the packet. This technique, which rely on the examination of *multiple fields* (at least two) is called *multi-field classification*. Multi-field classification, which is to be performed on every packet creates serious overhead to RSVP routers.

12. RSVP SCALABILITY

The overhead of packet classification is one of the drawback of RSVP and raises a scalability issue.

Another scalability difficulty lies in the fact that *states* -that is, a block of memory that stores static and dynamic information about the reservation- has to be created and maintained for every pair of sender and receiver that has to enjoy an improved service. If this may be appropriate for small scale Intranets with a limited number of concurrent RSVP reservations, the need to maintain states per individual flow is unsuitable for large scale Internets. This was the argument from opponents to RSVP, and this triggered the development of another, complementary technology called *DiffServ* that we see later.

13. SUMMARY OF RSVP DEFICIENCIES, AND WAYS FOR OVERCOMING THEM

We have seen that RSVP, though being an effective mechanism for reserving resources has three shortcomings:

1. **The per-flow states to maintain in routers**

   This entails a scalability problem for large scale Internets

2. **The multi-field classification on data packets**

   The overhead generated by the need to analyze every packet by examining multiple fields in order to determine the service class of that packet

3. **The instability of routes and the lack of quality of service-based routing**

   The fact that resources are reserved along routes and that we cannot guarantee the data traffic will follow these routes.

We see in the next sections how three rather recent technologies can be associated to RSVP so as to overcome most of those deficiencies. These technologies are:

- *RSVP aggregation*, to overcome problem 1
- *Diffserv*, to overcome problem 2
- *MPLS*, to overcome problem 3

14. RSVP AGGREGATION

RSVP Aggregation is a recent technique not yet stabilized at the time of this writing but very promising. The aim is to drastically reduce the number of states to be maintained in the network. This is a technology to be used within the core of the large scale Internet networks, not a technology for Intranets or limited scale Internets. The idea is to replace the reservations made per individual flow in the plain RSVP, by reservations made for aggregates of flows. Which aggregates?

Simply consider the core of a network and imagine that this core uses RSVP aggregation, that is, all the routers in the core understand the RSVP aggregation protocol. This core is made of a number of routers, some of them being *edge routers* -that is, having connection to the "non-RSVP aggregation" periphery- and others being *interior routers* only. The idea is that for any pair of edge routers, a *single* reservation will be made for all the flows that enter the core through one router of the pair and that exit the core through the other router of the pair. Thus, all those flows with the same *ingress* and *egress* router will share the same reserved path, and the amount of resources reserved over that path will match the sum of the individual requirements (not necessarily exactly).

As a result, if there are N edge routers surrounding the RSVP aggregation core, we will have (N^2–N) paths, therefore states, to maintain (remember, the reservation are unidirectional, thus we need two paths per pair in practice). This may be a big number but anyway lower than the total number of individual flows aggregated within the reserved paths.

15. DIFFSERV

Diffserv, which stands for *Differentiated Services*, is another recent technique aiming at overcoming the problem of heavy classification -that is the process for routers of knowing which service class a packet belongs to. The idea it to "mark" the packets with and indication of their priority in order to avoid having routers examining multiple fields. This mark is called a "*differentiated mark*", or *a Diffserv Code Point* (*DSCP*) and serves to map to a *differentiated treatment* to be applied to the packet. For a fast classification, the "mark" must be:

- of fixed length
- located at the beginning of the packet
- in a fixed position
- to be used as a direct pointer to find out what the differentiated treatment is to be.

The use of the mark is similar to that of RSVP aggregation: this is a technique which assumes that there is a core in the network which is "*Diffserv-capable*", that is, made of routers which understand the Diffserv marks and know how to exploit them for efficiently determining the packet priority. At the *edge* of this Diffserv core, the edge routers must be provisioned with the appropriate instructions to mark the packets (e.g. based on identification of flows such as source and destination addresses)

Therefore, by combining in the core of the network, RSVP aggregation and Diffserv marking, we can overcome two of the deficiencies of the plain RSVP:

- RSVP aggregation reduces drastically the number of reservations to maintain
- Diffserv removes the overhead of CPU-consuming classification by providing a simple, fast way of knowing the packet priority.

The final technology we briefly see, *MPLS*, will help us with the third difficulty, the route stability and the need for finding routes which also satisfy some quality of service constraints.

16. MPLS

MPLS stands for *Multi-Protocol Label Switching*. This is not a Quality of Service technique per se. This is a *forwarding* technology, a new approach to build network nodes which are neither pure IP routers nor pure switches, rather *hybrid* objects which try and combine the good points of both systems. The objective of MPLS is to build network nodes which can forward packets fast, hopefully at "hardware speed". More exactly, the technology provides a mechanism for making a fast *forwarding decision*, that is deciding to which outgoing link an incoming packet should be sent. To this end, MPLS uses a *logical reference*, called a *label*, which is inserted somewhere in the header of the packet. MPLS routers are provided with forwarding tables that map the incoming label to an outgoing link. The mechanism is analogous to the switching done in ATM or in X.25 switches.

MPLS, as RSVP aggregation and Diffserv, uses the *core* and *edge* principle. The MPLS technology is used within an "*MPLS-capable*" core, a mesh of MPLS routers. When entering the core, all packets taking the same route receive the same label, which is then used within the core by routers to take the forwarding decision, instead of the destination address as done outside the MPLS core.

This technique is not per se a solution to the remaining difficulty of the plain RSVP: the route stability and the Quality of Service-based Routing. However, it turns out that the paths followed by all the packet which carry the same label (called *LSPs*, *Label Switched Paths*) may be *constrained* to have a certain *stability* (for example, only change in case of failure), or to match certain *performance*

characteristics such as securing a certain bit rate or transit delay. The later facility, *called constraint-based routing* is nothing else than a form of Quality of Service-based Routing.

17. CONCLUSION

The combination of three techniques to be used in the core of large scale Internet:

- RSVP aggregation for reserving resources, yet limiting the number of states to maintain,
- Diffserv for a fast and easy classification of the data packets into service classes,
- and MPLS for providing appropriate route stability and Quality of Service-based Routing

is likely to provide in the future a solution which supports Quality of Service and is reasonably scalable.

18. BIBLIOGRAPHY

Understanding Network Multimedia, François Fluckiger, Prentice Hall,
ISBN: 0-13-190992-4

Quality of Service in IP Networks, Grenville Armitage, MacMillan
Technical Publishing, ISBN 1-57870-189-9

MPLS, Technology and Applications, Brice Davie and Yakov Rekhter, Morgan
Kaufann

Interconnection Second Edition, Radia Perlman, Addison-Weisley, ISBN:
0-201-63448-1

List of Students

J. Adamczewski, GSI, Darmstadt, Germany
M. Al-Turany, GSI, Darmstadt, Germany
V. Andreev, Joint Institute for Nuclear Research (JINR), Dubna, Moscow, Russia
A. Asimidis, University of Ioannina, Ioannina, Greece
Th. Berndt, Kischhoff-Institut für Physik, Heidelberg, Germany
Th. Boccali, Scuola Normale Superiore, Pisa, Italy
J.M. Bouche, CERN, Geneva, Switzerland
B. Boutherin, Institut des Sciences Nucléaires, Grenoble, France
C. Brusasco, GSI, Darmstadt, Germany
F. Calderini, CERN, Geneva, Switzerland
Ph. Canal, Fermilab, Batavia, USA
E. Cano, CERN, Geneva, Switzerland
V. Colin de Verdiere, CERN, Geneva, Switzerland
G. Daquino, INFN, Assergi (L'Aquila), Italy
O. Devroede, Vrije Universiteit Brussel, Etterbeek, Belgium
A. Ferrero, Università degli Studi di Torino, Torino, Italy
U. Fricke, DESY, Hamburg, Germany
G. Furano, Università di Roma "Tor Vergata", Rome, Italy
E. Garcia, CINVEsTAV-IPN, Mexico City, Mexico
J. Gutierrez Pacheco, CeCalCULA, Merida, Venezuela
S. Haug, University of Oslo, Oslo, Norway
M. Hemberger, GSI, Darmstadt, Germany
K. Holtz, Institut für Physik, Mainz, Germany
P. Homola, Faculty of Nuclear Sciences, Prague, Czech Republic
M.J. Jorda Garcia, CERN, Geneva, Switzerland
A. Kaczmarska, Niewodniczanski Institute of Nuclear Physics, Krakow, Poland
A. Kazarov, Petersburg Nuclear Physics Institute, Gatchina, Russia
M. Khan, Aligarh Muslim University, Aligarh, India
S. Kircher, University of Freiburg, Freiburg, Germany
A. Kugel, University of Mannheim, Mannheim, Germany
S. Lehti, Helsinki Institute of Physics, Helsinki, Finland
F. Lesser, University of Heidelberg, Heidelberg, Germany
N. Levitski, Institute for Theoretical and Experimental Physics, Moscow, Russia
O. Lopez, Laboratoire de Physique Corpusculaire de Caen, Caen, France
L. Lucio, CERN, Geneva, Switzerland
D. Mangeol, University of Nijmegen, Nijmegen, The Netherlands
M. Marczukajtis, Warsaw University of Technology, Warsaw, Poland
H. Matsumoto, University of Tokyo, Tokyo, Japan
C. Matthey, UCLA, Los Angeles, USA
O. Mitropoulos, University of Ioannina, Ioannina, Greece
M. Moles, CERN, Geneva, Switzerland
R. Mommsen, Laboratory for High Energy Physics, Bern, Switzerland

R. Mordente, National Institute of Nuclear Physics (INFN), Napoli, Italy
J. Moscicki, CERN, Geneva, Switzerland
M. Musy, University of Rome "La Sapienza", Rome, Italy
K. Nawrocki, Soltan's Institute for Nuclear Studies, Warsaw, Poland
P. Nilsson, Lund University, Lund, Sweden
A. Oh. CERN, Geneva, Switzerland
I. Palshin, Bogolyubov Institute of Theoretical Physics, Kiev, Ukraine
K. Petrov, Institute for Theoretical Physics, Kiev, Ukraine
I. Pivovarov, Budker Institute of Nuclear Physics, Novosibirsk, Russia
J.E. Pradas Simon, CERN, Geneva, Switzerland
D. Prokofiev, Petersburg Nuclear Physics Institute, St.Petersburg, Russia
E. Roux, CERN, Geneva, Switzerland
P. Saiz, CERN, Geneva, Switzerland
M. Sanchez, Tufts University, Medford, USA
M. Sapinski, H.Niewodniczanski Institute of Nuclear Physics, Krakow, Poland
W. Schleifer, CERN, Geneva, Switzerland
K. Schossmaier, CERN, Geneva, Switzerland
A. Sciabà, Istituto Nazionale di Fisica Nucleare, Pisa, Italy
K. Sigerud, CERN, Geneva, Switzerland
C. Soler, CDTI, Madrid, Spain
G. Troubnikov, Joint Institute for Nuclear Research, Dubna, Moscow, Russia
V. Tsulaia, E.Andronikashvili, Institute of Physics, Georgian Academy of Science, Tbilisi, Georgia
S. Udriot, ETHZ - Lab. für Hochenergiephysik, Zürich, Switzerland
I. Ueda, ICEPP, University of Tokyo, Tokyo, Japan
V.M. Vagnoni, Dipartimento di Fisica dell'Università, Bologna, Italy
J. Valenta, Faculty of Nuclear Sciences and Physical Engineering, Prague, Czech Republic
P. Vanlaer, IIHE/ULB, Brussels, Belgium
S. White, Fermi National Accelerator Laboratory, Batavia, USA
M. Winkler, Institute for High Energy Physics, Vienna, Austria
K. Wrona, University of Mining and Metallurgy, Krakow, Poland
Z. Xie, Istituto Nazionale di Fisica Nucleare, S. Piero a Grado, Pisa, Italy
C. Zacharatou Jarlskog, Lund University, Lund, Sweden

List of Students

J. Adamczewski, GSI, Darmstadt, Germany
M. Al-Turany, GSI, Darmstadt, Germany
V. Andreev, Joint Institute for Nuclear Research (JINR), Dubna, Moscow, Russia
A. Asimidis, University of Ioannina, Ioannina, Greece
Th. Berndt, Kischhoff-Institut für Physik, Heidelberg, Germany
Th. Boccali, Scuola Normale Superiore, Pisa, Italy
J.M. Bouche, CERN, Geneva, Switzerland
B. Boutherin, Institut des Sciences Nucléaires, Grenoble, France
C. Brusasco, GSI, Darmstadt, Germany
F. Calderini, CERN, Geneva, Switzerland
Ph. Canal, Fermilab, Batavia, USA
E. Cano, CERN, Geneva, Switzerland
V. Colin de Verdiere, CERN, Geneva, Switzerland
G. Daquino, INFN, Assergi (L'Aquila), Italy
O. Devroede, Vrije Universiteit Brussel, Etterbeek, Belgium
A. Ferrero, Università degli Studi di Torino, Torino, Italy
U. Fricke, DESY, Hamburg, Germany
G. Furano, Università di Roma "Tor Vergata", Rome, Italy
E. Garcia, CINVEsTAV-IPN, Mexico City, Mexico
J. Gutierrez Pacheco, CeCalCULA, Merida, Venezuela
S. Haug, University of Oslo, Oslo, Norway
M. Hemberger, GSI, Darmstadt, Germany
K. Holtz, Institut für Physik, Mainz, Germany
P. Homola, Faculty of Nuclear Sciences, Prague, Czech Republic
M.J. Jorda Garcia, CERN, Geneva, Switzerland
A. Kaczmarska, Niewodniczanski Institute of Nuclear Physics, Krakow, Poland
A. Kazarov, Petersburg Nuclear Physics Institute, Gatchina, Russia
M. Khan, Aligarh Muslim University, Aligarh, India
S. Kircher, University of Freiburg, Freiburg, Germany
A. Kugel, University of Mannheim, Mannheim, Germany
S. Lehti, Helsinki Institute of Physics, Helsinki, Finland
F. Lesser, University of Heidelberg, Heidelberg, Germany
N. Levitski, Institute for Theoretical and Experimental Physics, Moscow, Russia
O. Lopez, Laboratoire de Physique Corpusculaire de Caen, Caen, France
L. Lucio, CERN, Geneva, Switzerland
D. Mangeol, University of Nijmegen, Nijmegen, The Netherlands
M. Marczukajtis, Warsaw University of Technology, Warsaw, Poland
H. Matsumoto, University of Tokyo, Tokyo, Japan
C. Matthey, UCLA, Los Angeles, USA
O. Mitropoulos, University of Ioannina, Ioannina, Greece
M. Moles, CERN, Geneva, Switzerland
R. Mommsen, Laboratory for High Energy Physics, Bern, Switzerland

R. Mordente, National Institute of Nuclear Physics (INFN), Napoli, Italy
J. Moscicki, CERN, Geneva, Switzerland
M. Musy, University of Rome "La Sapienza", Rome, Italy
K. Nawrocki, Soltan's Institute for Nuclear Studies, Warsaw, Poland
P. Nilsson, Lund University, Lund, Sweden
A. Oh. CERN, Geneva, Switzerland
I. Palshin, Bogolyubov Institute of Theoretical Physics, Kiev, Ukraine
K. Petrov, Institute for Theoretical Physics, Kiev, Ukraine
I. Pivovarov, Budker Institute of Nuclear Physics, Novosibirsk, Russia
J.E. Pradas Simon, CERN, Geneva, Switzerland
D. Prokofiev, Petersburg Nuclear Physics Institute, St.Petersburg, Russia
E. Roux, CERN, Geneva, Switzerland
P. Saiz, CERN, Geneva, Switzerland
M. Sanchez, Tufts University, Medford, USA
M. Sapinski, H.Niewodniczanski Institute of Nuclear Physics, Krakow, Poland
W. Schleifer, CERN, Geneva, Switzerland
K. Schossmaier, CERN, Geneva, Switzerland
A. Sciabà, Istituto Nazionale di Fisica Nucleare, Pisa, Italy
K. Sigerud, CERN, Geneva, Switzerland
C. Soler, CDTI, Madrid, Spain
G. Troubnikov, Joint Institute for Nuclear Research, Dubna, Moscow, Russia
V. Tsulaia, E.Andronikashvili, Institute of Physics, Georgian Academy of Science, Tbilisi, Georgia
S. Udriot, ETHZ - Lab. für Hochenergiephysik, Zürich, Switzerland
I. Ueda, ICEPP, University of Tokyo, Tokyo, Japan
V.M. Vagnoni, Dipartimento di Fisica dell'Università, Bologna, Italy
J. Valenta, Faculty of Nuclear Sciences and Physical Engineering, Prague, Czech Republic
P. Vanlaer, IIHE/ULB, Brussels, Belgium
S. White, Fermi National Accelerator Laboratory, Batavia, USA
M. Winkler, Institute for High Energy Physics, Vienna, Austria
K. Wrona, University of Mining and Metallurgy, Krakow, Poland
Z. Xie, Istituto Nazionale di Fisica Nucleare, S. Piero a Grado, Pisa, Italy
C. Zacharatou Jarlskog, Lund University, Lund, Sweden