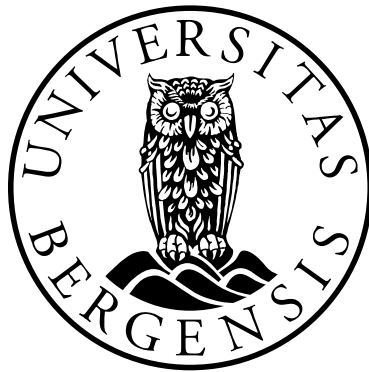


Autonomic Operation of a Large High Performance On-Line Compute Cluster

Øystein S. Haaland



Dissertation for the degree of philosophiae doctor (PhD)
at the University of Bergen

2015

Dissertation date: 10.12.2015

Abstract

A High-Level Trigger (HLT) system is composed of both hardware and software. Designing the physical layout of the cluster mainly concerns the hardware; node distribution, network layout, estimation of power requirements, defining hardware properties of the nodes and so on. These are to a great extent derived from the processing topology to be used in the software application, which in turn is chosen based on the nature of the data to be processed.

In a project the size of HLT, it is challenging to predict the specifications of the hardware to be bought in the future, while accounting for requirements that may change as the project matures over time. The first part of the thesis is a review and an evaluation of the stages from early design to a fully operational HLT, presented from an instrumentation and software engineering point of view.

Differences between computational science and software engineering became apparent early on. The existing literature on the topic helps to understand the observations, and from this understanding, suggestions for possible improvements that could benefit similar projects in the future are made. Suitable concepts, technology and practices have been identified by researching current trends and looking to other relevant fields of study.

The solutions that could be implemented and evaluated during the course of the work on the thesis, are verified in prototypes. Autonomic computing has been an important inspiration as well as the management specifications from the Distributed Management Task Force (DMTF). A general observation is that there seems to be much that potentially could be learned from software engineering, a field that has been working on large scale software systems for a long time. Although one must be cautious and critical in what is adopted, since not everything will apply. It goes without saying, that scientific computing has its own contributions to the generic computing field. The prototypes are described in the last part of the thesis, where also the acceptance criteria and other results can be found.

Acknowledgments

Finally being able to conclude the work and involvement in ALICE HLT that started so many years ago is a peculiar feeling. Spending almost three years - in total - stationed at CERN and helping out with commissioning has been a very interesting and exiting time in the anticipation of LHC startup. The work has been challenging and at times stressful, but also very rewarding. I believe what I have learned at CERN as part of the on-site HLT team can hardly be learned anywhere else.

There are many of which I am in great debt for support and encouragement. First of all, I would like to thank Prof. Dieter Röhrich and Prof. Håvard Helstrup for having been given the opportunity to take part in such a great effort in HEP history first hand. Furthermore I would like to express my gratitude for the patience you have shown during the last three years, as I have gradually worked towards completing the programming part and the writing of the this thesis on the side of an unrelated full time job. Throughout the process, your guidance has been much appreciated when the road forward has been unclear, and the feedback has always been prompt and thorough.

During the time at CERN, I must in particular mention Jochen Thäder and Stefan Kirsch. Together with Jochen I was one of the first from the collaboration working on-site for the HLT during the early commissioning. We spent countless hours together preparing the HLT cluster. Either in the in the counting room at the ALICE pit or in the office we shared at the main site. It was a pleasure working together. In the same office, was Stefan Kirsch, with whom I also had the privilege of sharing a flat for two years, and thereby also the everyday ups and downs of being at CERN during commissioning. Your company during the stay was much appreciated.

Later on, the on-site HLT team, grew to include Artur Szostak and Torsten Alt. Not only colleagues, but also generous neighbors. Many topics have been discussed on their balcony over German beer while staring at the Jura mountains. Timm Steinbeck, Matthias Richter, Timo Breitne and Sergey Gorbunov have all been inspiring in their engagement, development and stabilization of the core HLT application.

Also from the time at CERN, I have many good memories from spending time with Øystein Djuvsland, Kyrre Skjerdal, Alex Kastanas and Therese Sjørnsen, exercising with Kenneth Aamodt, Svein Lindal and Per Thomas Hille, from sharing lunch with Magnus Mager, Jens Steckert and Christian Lippmann and many more, as well as frequenting the ALICE control room with Indranil Das, Arshad Masoodi, and Dinesh Ram.

A special nod goes to fellow system administrators of the HLT cluster; Pierre Zelnicek and Olav Smørholm and a thank you to Marian Hermann, Jochen Ulrich and Camilo Lara for the collaboration on the paper I got to present at the Systems and Virtualization Management workshop in 2011, as well as to the rest of the SysMES group (Stefan Böttger).

I have greatly appreciated the company of my colleagues at the Department of Physics and Technology in Bergen: Kelly Kanaki, Dag Toppe Larsen, Sebastian Bablok, Camilla H. Stokkevåg, Hege Erdal, Kristian Ytre-Hauge, Lijiao Liu, Meidana Huang and Dominik Fehlker. Sedat Altinpinar, has been an excellent office mate during the odd hours I have come by the institute (thanks for the Turkish delights!) and with Boris Wagner I have had the pleasure to discuss many of the topics that relates closely to the work in my thesis.

I'm forever grateful for the down-to-earth support from my family, in particular my parents Berit and Hallvard, who have always encouraged me to find my own ways and shown me how to appreciate life by their conduct and actions, as much as by anything else. But also from my wonderful siblings Anders and Inga, who brings me much joy and amusement whenever we meet.

Finally, Lone, my partner and companion in life, has been with me on this journey from the beginning. You were there with a delicious surprise dinner and bubbles when I delivered my masters in the middle of the night, and reportedly looked "a little bit crazy". You traveled relentlessly to Geneva to visit during the years we spent apart while I were at CERN. You have been patient with me and selflessly supporting me while I have spent after-work hours in finishing this thesis and you are still here now by the end of this journey. And if all goes well, we are about to start a new journey together, for which I'm very excited. I can only say: thank you!

Contents

1. Introduction	15
2. Software engineering and scientific computing	17
2.1. General concepts	17
2.2. Chasms and gridlocks	17
3. A Large Ion Collider Experiment	20
3.1. Detectors	21
3.2. Trigger	22
3.3. Data Acquisition	23
3.4. Control systems	25
3.5. Offline Grid analysis	26
4. ALICE High Level Trigger	27
4.1. Triggering in High Energy Physics	27
4.2. Requirements	28
4.2.1. Functional requirements	29
4.2.2. Non-functional requirements	29
4.3. Hardware	31
4.3.1. Cluster nodes	33
4.3.2. Input and output (H-RORC)	34
4.3.3. Processing	35
4.3.4. Storage	35
4.3.5. Interconnect	36
4.3.6. Cluster management (CHARM card)	37
4.4. Software	38
4.4.1. Software system anatomy	39
4.4.2. The physics application	41
4.4.2.1. Analysis framework	43
4.4.2.2. Data transport framework	43
4.4.2.3. Chain configuration software.	43
4.4.2.4. Interfaces	44
4.4.3. Software environment	45
4.4.4. Cluster management software	46
4.4.4.1. The SysMES Framework	47
4.4.4.2. Infologger	47
4.4.4.3. Inventory database	48

5. Experience and performance	49
5.1. Installation, Commissioning and Upgrades	49
5.1.1. Node installation	49
5.1.2. Network	50
5.1.3. Storage and system services	50
5.1.4. Clustering and virtualization	50
5.1.5. Big design up front	51
5.2. Implementation and development	52
5.2.1. Developer infrastructure	52
5.2.2. Debugging and testing	53
5.2.3. Build configuration	53
5.2.4. Shell scripts	54
5.3. Scope and methodology	54
5.4. Physics performance	55
5.5. Operational performance	56
5.5.1. Mean time between failures	56
5.5.2. Recorded log messages	60
5.6. Effort estimation and development performance	61
5.6.1. Source lines of code and the Constructive Cost Model	62
5.6.2. Developer activity	63
5.6.3. HPC productivity	70
5.7. Software quality	70
5.7.1. Technical debt	70
5.7.2. Code coverage	71
5.7.3. Code smells: code duplication	72
5.8. Summary	72
6. Possible improvements	75
6.1. The autonomic computing initiative	75
6.2. Current trends and relevant concepts	77
6.2.1. Increased performance of dynamic (interpreted) languages	77
6.2.2. Growth of Free and Open Source Software (FOSS)	79
6.2.3. Clusters, the Grid and Cloud	79
6.2.4. Loose coupling and the resurgence of REST	80
6.2.5. NoSQL - unstructured and simplified for scalability	82
6.2.6. Asynchronous and event driven	82
6.2.7. The high-level, resource-elastic and friction-less future	82
6.3. Middleware and distributed management	83
6.3.1. Object-Relational Mapping	84
6.3.2. Inter-process communication/remote API access	84
6.3.3. Data serialization	85
6.3.4. Service discovery	85
6.3.5. DMTF management standards and the WBEM stack	86

Contents

6.4.	Developer tooling	86
6.4.1.	Source code management	87
6.4.2.	Task automation	87
6.4.3.	Build configuration	88
6.4.4.	Testing and continuous integration	89
6.4.5.	Packaging and deployment	90
6.5.	Software engineering practices	90
6.5.1.	Code review/inspection	90
6.5.2.	Iterative development	90
6.5.3.	Code reuse	91
6.5.4.	Test Driven Development	91
6.5.5.	Executable documentation and literal programming	92
6.5.6.	Collective Code Ownership	93
6.5.7.	Model driven engineering (code generation)	93
6.6.	Monitoring, logging and debugging	93
6.6.1.	Live monitoring and logging	94
6.6.2.	Metrics gathering and visualization	95
6.6.3.	Debugging large and complex systems	95
6.7.	Summary	96
7.	Design and implementation	98
7.1.	General requirements	98
7.2.	Existing software	99
7.3.	Cluster API	100
7.3.1.	Use cases and requirements	101
7.3.2.	Design	103
7.3.3.	Core concepts	104
7.3.4.	Implementation	106
7.3.4.1.	Process management	106
7.3.4.2.	Local service reflection	107
7.3.4.3.	Remote service discovery	108
7.3.4.4.	Eventloop	111
7.3.4.5.	Method dispatch	112
7.3.5.	Core API	114
7.3.6.	Future improvements	115
7.4.	Inventory database	115
7.4.1.	Use cases and requirements	116
7.4.2.	Design	117
7.4.3.	Remote Persistent Objects	118
7.4.4.	Implementation	119
7.5.	Presentation and user interaction	125
7.6.	Rounding out the autonomic aspects	132

Contents

8. Results and testing	136
8.1. Implementation status	136
8.1.1. Clusterapi	136
8.1.2. Cimpv	136
8.1.3. Hwdiscover	137
8.1.4. Nge	137
8.2. Developer tooling, metrics and artifacts	138
8.3. Full scenario integrated acceptance test	139
8.4. Performance and scalability	145
8.5. Evaluation of suggestions and validity of guidelines	146
9. Conclusion and outlook	147
A. SLOC count	150
A.1. Internal packages	150
A.2. External packages	151
A.3. Programming languages	151
B. Git statistics	152
C. Use cases	154
C.1. Actors	154
C.2. Chain operation	155
C.3. Resource management	156
C.4. Cluster administration/Service management	157
C.5. Data stream connection	158
C.6. Statistics retrieval	159
C.7. RORC control	160
C.8. SysMES	161
C.9. Clusterbus use cases	161
D. Service discovery of data sinks	162
E. Log data mining	163
F. Software configuration/build system	167
F.1. The Software packages of HLT	167
F.2. CMake example	168
F.3. Full cycle testing of infologger	168
G. Acceptance test	169
G.1. Test script	169
G.2. Test log	175
Bibliography	180

List of Figures

3.1. Setup of the A Large Ion Collider Experiment (ALICE) detector system. . .	20
3.2. 3D view of the TPC field cage	21
3.3. Schematic view of front end and readout electronics.	22
3.4. Overview of the data flow in ALICE.	24
3.5. Data Aquisition (DAQ)-High-Level Trigger (HLT) data flow.	25
4.1. The six architectural layers of the HLT	30
4.2. Overview of the counting rooms and the experiment in the ALICE pit. . .	32
4.3. The HLT counting room CR2	33
4.4. HLT - Read-Out Receiver Card (H-RORC) data pump and Field-programmable gate array (FPGA) coprocessor.	35
4.5. Picture of counting room 2.	37
4.6. Software components and actions involved in chain operation.	42
4.7. HLT interfaces to other systems of ALICE.	44
4.8. The event display showing one of the first pp collisions at 7TeV [39]. . . .	45
4.9. DATE infoLogger architecture.	47
5.1. Amount of run-time - in accumulated seconds - with HLT participation per week.	57
5.2. Percentage of run-time with HLT participation per week.	58
5.3. End of run reason. All runs to the left and failed runs to the right. . . .	59
5.4. Start of run failure to the left and subsystem failures to the right. The labels are extracted directly from the data. In the right figure TR means trigger, HL means HLT, FE means FERRO and DC means DCS.	59
5.5. SOR_failure for HLT only.	60
5.6. Commit activity by year (top) and month (bottom) as produced by gitstats [91]. Shows number of commits over time for the whole hlt-alice repository. . .	64
5.7. RUP phases and activities.	65
5.8. Diagram of commit activity per project over time, showing run-critical projects. One dot represents a commit in time for the developer listed to the left.	67
5.9. Diagram of commit activity per project over time, showing projects related to operation. One dot represents a commit in time for the developer listed to the left.	68
5.10. Diagram of commit activity per project over time, showing the remaining projects in the main HLT repository. One dot represents a commit in time for the developer listed to the left.	69

List of Figures

7.1. Core use cases for clusterapi.	102
7.2. Overview of process control and XML-RPC traffic in clusterapi. In this case, two services have been started; hwds and hwdc. Their methods have been exposed through the clusterapi and supervisord manage the processes.	107
7.3. Supervisor and clusterapi interaction. Here shown when starting a third process, calling a method on the started service and finally stopping the same process.	108
7.4. Sequence diagram of two clusterapi instances starting up on two hosts, using service discovery. They act as peers, having symmetric sequences. The announcement and browse steps sets everything up so that the following steps are driven by events.	110
7.5. Service discovery and the supervisord listener shares the event loop with the XML-RPC proxy. Service discovery information is communicated over a D-Bus interface, while supervisor communicates over a standard input/output line-based protocol.	111
7.6. Class diagram of clusterapi.	111
7.7. Method dispatch example for two consecutive method calls. The increased response time for host 1 causes a reordering so that the next call to method 1 goes to host 2.	114
7.8. Use cases for the inventory database.	117
7.9. Cimpv generation.	120
7.10. Implemented CIM classes.	121
7.11. Sequence diagram showing hwdserver and hwdclient communication when integrated in clusterapi. Actions are triggered by a third process, Nge.	123
7.12. Conceptual overview of hwddiscover and interaction with related systems.	124
7.13. Class diagram for nge parser.	126
7.14. Interaction between nge and clusterapi.	130
7.15. High level overview of command and information flow in clusterapi, inventory database and nge. Communication between clusterapi instances are shown as well as information flow for the inventory database instances.	130
7.16. Screen shot of the Nge inventory database.	132
7.17. Example of how a verification language could look.	135
8.1. Screenshots showing status information right after Nge has been started.	142
8.2. Screenshot showing a new node appearing in the supervisor dashboard after the second clusterapi instance have been started.	143
8.3. Screenshots showing how status has changed after the hwd client has been started on the second clusterapi instance.	144
C.1. Overview of actors.	155
C.2. Use case diagram of chain operations.	156
C.3. Use case diagram for resource management.	157
C.4. Use case diagram for cluster administration.	158
C.5. Use case diagram for data stream connections.	159

List of Figures

C.6. Use case diagram for statistics retrieval.	160
C.7. Use case diagram for RORC control.	160
C.8. Use case diagram for SyMES operations.	161
E.1. Total number of log messages per day	164
E.2. Total number of log messages per week	165

List of Tables

4.1.	Repositories containing software used by HLT. The involvement of HLT is indicated in parenthesis next to the project name.	39
4.2.	Software packages from the hlt-alice and aliroot repositories with a short description. The packages necessary for HLT operation are those with critical set to True. Projects labels are composed of <reponame>:<projectpath>, where project path can be a sub-directory.	40
5.1.	COCOMO variants of the full hlt-alice repository together with an entry where all subprojects are summed up. The numbers marked with an “*” was not produced directly by sloccount, but summed up over the individual project results. COCOMO numbers for external packages used in HLT can be found in the appendix A.2.	63
5.2.	Comparing COCOMO numbers (right of vertical line) to commit activity (left of vertical line). Months for commit activity is the number of months between the first and last registered commit. Changes is the number of lines that has been changed in the commits.	66
5.3.	Software quality metrics for HLT software.	71
8.1.	Implemented and envisioned future method call types in Clusterapi. . . .	136
8.2.	Tables showing the latest unit test, coverage and lint results for cimpy, hwdiscover, clusterapi and nge. Extracted on two occasions, the first one on 14.04.2012 and the second during june 2014.	139
8.3.	A table showing number of nodes, time to complete, CPU load, memory usage and file size for the inventory database in different testing environments when collecting all inventory information. All measurements are performed with the top command line tool	145
A.1.	COCOMO numbers for individual sub-projects in hlt-alice, showing number of developers, person months, scheduled months. Cost is given in US dollars.	150
A.2.	SLOC numbers for external packages used in HLT.	151
A.3.	Statistics of programming languages used in HLT. Internally developed packages to the left and external packages to the right. The tables shows actual number of lines per programming language and percentage of total number of lines.	151

List of Tables

B.1. Key numbers showing the timespan and total number of; commits, authors and lines changed for the individual sub-projects of the hlt-alice repository. More information about the tools used can be found here in the appendix B.	153
E.1. Overview of the data sets. MySQL[244] is used with the innodb engine and the database size is measured by the size of the message.MYD file . . .	163
F.1. Build systems in HLT software. (see docs/SoftwareCatalog/software_catalog.lyx). Where it says make, pure make is meant, not with any autotools assistance. In ROOT Bash is used configuration and make for building. Lately CMake support have been added. Infologger was originally only pure make files. With the adoption to HLT, CMake files was added	167

List of Algorithms

7.1. Demonstration code for envisioned clusterapi.	105
7.2. Data structure in JSON representation for method dispatch.	113
7.3. Example of calling python code from for-each tags.	128
7.4. Example of calling python code from link tags.	129
7.5. A page excerpt from a Nge site definition that will fetch BIOS information for all nodes in a hwdiscover store and present it as can be seen in figure 7.16.	131
7.6. Example of fictitious command-line interaction. Similar operation should also be available from a web interface. In both cases, the interaction should be done towards the same service interface.	134
8.1. General query for hwdiscover ORM store.	138
E.1. Script that queries the infologger database for basic information.	166

1. Introduction

High performance computing have become indispensable tools for problem solvers of today. Domains as diverse as medicine, finance, meteorology, oil and physics all use these number crunchers like never before to acquire new insight [1]. While the human brain is a unique and marvelous thing - the most capable and versatile device yet observed - there are some tasks where it comes short when compared to machines. Certain types of problems are of such a nature that they are most easily solved by performing calculations on large data sets. Doing this in a concise, predictable and efficient manner is where the machine excels. In fact, many of most difficult and pressing problems might only be solvable with the help of supercomputers [1].

The trend in computer history has gone from a powerful central machine with thin clients to thicker clients with relatively powerful servers. This trend continues in the history of supercomputers, where these machines used to be built as big mainframes, but as time have passed, they are increasingly being built as clusters composed of Commercial off-the-shelf (COTS) computers with fast interconnects.

Driven by the consumer market, the high volume production of such machines make them cheap and widely available. With the performance increase consumer hardware have seen over the last decade and the improved network connectivity, compute clusters have become the preferred architecture for supercomputers [2].

The cost of using such a composed architecture, is increased complexity for the programmer who wants to formulate his or her problem [1]. Parallelism, pipelining, state synchronization, message-passing are just some examples of what one might have to get familiar with when working in High Performance Computing (HPC). After many years of research and development, the HPC community is still faced with the challenge of managing such a machine and presenting it as a unified system to its users in a way that makes it easy to express the problems to be solved.

Throughout history, scientists have been used to building their own instruments for performing experiments. But the complexity of the machines that are being built today, requires expert knowledge outside the scientific application domain. In this case knowledge related to the instrument itself, from computer science related fields, such as informatics, software engineering and cluster management.

The work presented here, concerns a very specialized compute cluster used for online event building, event filtering and data compression in High Energy Physics (HEP). It is part of the ALICE online system, one of the experiments along the Large Hadron Collider (LHC) accelerator at European Organization for Nuclear Research (CERN).

The main task of this dissertation is to undertake a critical performance review of the ALICE HLT project. To look at how computing requirements were derived from physics requirements (section 4.1) and how implementation was executed according to

1. Introduction

these (sections 5.1 and 5.2). The angle of approach will therefore be in the intersection between physics instrumentation and computer science. Special attention will be paid to how changes in a fast-moving IT-industry have influenced requirements in the HLT project and how the project has coped with such influence.

The discussion about implementation and performance will be followed up with suggestions for improvements. The potential for improvement in a project like HLT spans all the way from the large scope of project management to the smallest details of software development practices. It can include suggestions for new designs and implementations of missing infrastructure, all depending on the outcome of studies to be conducted. Relevant improvements will be evaluated and conclusions will be made about their feasibility. The conclusions will be summed up as a set of recommendations that will be presented as a part of the result from this work.

A few of these concepts and designs that are most relevant to the perspective of this thesis, will be sought verified or falsified in prototype implementations tested under realistic conditions. The outcome of this testing will form the second part of the presented result.

One component has already been identified as a desired addition to the HLT cluster; an inventory database. Apart from keeping track of the inventory, it is the precondition of resource management, an important service for making the the HLT more autonomous and fault-tolerant. The implementation of such an inventory database will be used as a verification of suggested software development practices as well as for actual design. A second implementation, also targeting the same system, will be described for comparison of approaches, methodologies and software development metrics.

The next chapter presents the general problem domain of software engineering and scientific computing. Then follows a description of the context, that is LHC and ALICE, in which HLT operates. The fourth chapter will describe HLT as it exists today in some detail, including software, hardware and infrastructure, and there will be sections on requirements and architectural design concepts. The fifth chapter is concerned with the stages of development in HLT and how the design and implementation changed over the years in order to accommodate for changing requirements. The focus will be on the experiences and challenges throughout the process. The remaining sections of the chapter will discuss the performance of various aspects of HLT.

The sixth chapter makes suggestions for improvements to HLT. About technology that can be deployed and practices that can be used to improve upon the current project, and for similar projects in the future. Chapter seven proposes a design for a common architecture for (first and foremost non-core) HLT applications as well as design of other tools that are currently missing in HLT. Prototypes of these designs are also presented with the intention of verifying how suitable the suggestions are in the HLT cluster and to evaluate them for future use. Chapter eight will list any results so far and finally there will be a conclusive chapter that also takes a look forward.

2. Software engineering and scientific computing

2.1. General concepts

Many science fields are involved simultaneously in building a system like the HLT. In addition to physics and instrumentation, there is an array of computer science related fields also being applied in today's experiments. The body of academic literature to draw from is therefore large. Important sources includes Association for Computing Machinery [3] (ACM), that covers many computer science sub-fields, among them software engineering [4], in their special interest groups. Then there is the Journal of Computing in Science & Engineering [5], which has held a workshop dedicated to Software engineering [6].

High Performance Computing is covered in the Journal of High Performance Computing Applications [7], supercomputing by the supercomputing conference series [8], and the conference most specialized in HEP and generally most relevant for HLT is Computing in High Energy and Nuclear Physics (CHEP).

Other relevant sources for this dissertation is the ACM International Conference on Autonomic Computing, now called The ACM Cloud and Autonomic Computing Conference. Lastly, there is the System and Virtualization Management conference held by the Distributed Management Task Force (DMTF).

2.2. Chasms and gridlocks

While Software Engineering (SE) has established itself as a field of study that the industry can rely on for improving the ways in which we create software, it is not given that the experience gained in the general software domain will be useful in the individual scientific domains. Given the potential gains that can be achieved through software engineering though, it is valuable to know which solutions are, and which are not, applicable to scientific computing.

After having discovered what has been described as a chasm between SE researchers and computational scientists in the field [9], and later observed that there seem to be a productivity gridlock in computational sciences [10], it was eventually acknowledged that software development in scientific computing is markedly different from other domains.

A study initiated by Sun [10], points out that properties that are important for scientific computing, like performance, real-time constraints, hardware cost and portability, holds much less significance in the software engineering community, who rather favors maintainable code, robust programming languages and higher abstraction levels. With such conflicting needs, there is no wonder why computational scientists are frustrated

2. *Software engineering and scientific computing*

with all the “fads” being imposed on them from the SE academics. Likewise, it is no surprise that software engineering sees scientific computing as being arcane and primitive in their ways of building software.

It has been suggested that the arrival of this state of mind has been due to a fundamental change in how application programming is viewed in the academic discussion [11]. Computer science was from the beginning primarily concerned with the application - the problem - itself, but not far into its own history, this focus changed, as one predicted methods would soon be found that could deliver automatic solutions to a set of problems. Finding these generic methods that could be used to derive specific solutions automatically therefore became the main occupancy of the researchers, neglecting to a certain extent the problem domain.

This insistence on generic applicability, not taking into account the needs of the specific domain, is considered to be the root cause of the chasm between practitioners and academics in today's scientific computing and software engineering communities [9]. While focusing on domain-independent solutions has been fruitful over the years, it seems to have run its course in providing further progress, and maybe is the frame of view it imposes prohibitive for moving research forward. Having realized this, there is a growing interest from the SE community in trying to understand the characteristics of the computational-heavy sciences, like HPC [12, 13].

Going off on a tangent, it is interesting and maybe sobering to be reminded of the assessment about essential (creating the conceptual structure and abstract software entity) and accidental (representing abstract concepts in programming language and producing an executable product within given constraints) efforts in the Mythical Man-Month [14]. The chapter on “No Silver bullet” concludes that much has been achieved in reducing accidental efforts - the transition from punch cards to interactive systems can serve as an example - and that these improvements have been in the order of a magnitude or more, the kind of scale that justifies being called fundamental changes. But that there is not much more to be gained; there will be no change in the future that can be compared to what has already been achieved, in continuing this vain.

One would now have to turn to essential efforts in the complexity of software engineering to progress, accepting that large improvements come in small increments accumulated over long periods of time - there are no silver bullets left.

Considering the influence this book has had and its message, it is probably fair to assume that it would be one of the voices that has spurred the direction that has been taken. At the same time, its advice on setting realistic expectations and prepare for slow, but steadfast progress is maybe a valuable input from the past in today's hunt for higher programmer productivity.

Returning to SE's increased interest in scientific computing, the multidisciplinary demands in HPC have been identified as one of the main challenges for improved programmer productivity [10]; not only do you need to be a domain expert in your scientific branch, but also an expert in software optimization (often for a given hardware architecture) and scalability (making the software run in a parallel to improve performance).

Related to this is another challenge: the accidental efforts resulting from the extra - usually manual - work needed for achieving the high computational performance sought

2. *Software engineering and scientific computing*

in HPC. These efforts do not directly contribute to solving the problem at hand, but are rather tied to the chosen tooling and machinery used to describe and process the problem [10].

With a fresh change of mind-set, where it is acknowledged that certain methods only works for certain domains, one direction of effort has been to try, as a first step, to create a taxonomy of methods and a taxonomy of application domains. With these in hand, the following step would be to try to match methods to domains, to create a instrument for advising domain experts of suitable methods for their field [11, 15].

An important part of this work will be to find the appropriate decomposition of concepts that will result in the least possible overlap when mapping between the two taxonomies. One can expect that the taxonomies and mappings to evolve together as research moves forward, although it may take time, as even defining the application taxonomy has proven to be challenging [16].

Along another axis, we have to remember that software engineering is still a young field, we have yet to treat it like a proper science, with the strict demand for evidence and applying the scientific (scrutiny) tools available. Adding that there currently is a lot of handwaving, claiming it is difficult to know much about software development, while at the same time observing that little effort is made in trying to move the field out of its infancy and into maturity [17, 18].

With these initiatives finally happening, it is discouraging to learn about the reflections on software engineering by long time members of the HEP community [19], that to a large extent confirms and exemplifies recent findings; that of being entrenched in ones own side of things and being unaware of recent research on software engineering in scientific computing.

It seems nevertheless that the two sides agrees, that much can be gained by collaborating closer, but also realizing that effort is required from both sides. Software engineers have to become better at addressing the scientific community by taking what they know today from other domains, take it apart and try to re-apply the knowledge to the specific scientific domains. The scientific computing communities on their hand must become more open to change, and use the software engineering for what its supposed to be; an aid in improving the software development for the various application domains that heavily relies on software for their daily business.

Making software engineering part of the training in computationally-heavy sciences is another suggestion [20]. It is not realistic to expect scientists to be experts in more than one field, so the training would have to be practical and focusing on the basic tools and techniques that are the most important. The attempts at educating scientists in (practical, essential) software engineering have so far been encouraging and resulted in verifiable improvements (around 20%) [21].

Trends, technology, tools and practices relevant in this respect are the topic of chapter 6.

3. A Large Ion Collider Experiment

ALICE is a general-purpose heavy-ion detector system designed to study strongly interacting matter and the quark-gluon plasma produced in nucleus-nucleus collisions (Pb-Pb) [22]. ALICE is composed of 18 sub-detectors. From the information produced, events can be reconstructed and stored. The analysis itself is performed on large Grid [23] systems after the data has been collected.

While having been designed for nucleus-nucleus collisions, ALICE is also capable of detecting proton-proton interactions. LHC delivers proton-proton collisions most of the year, while approximately one month, towards the end, is dedicated to Pb-Pb and p-Pb runs.

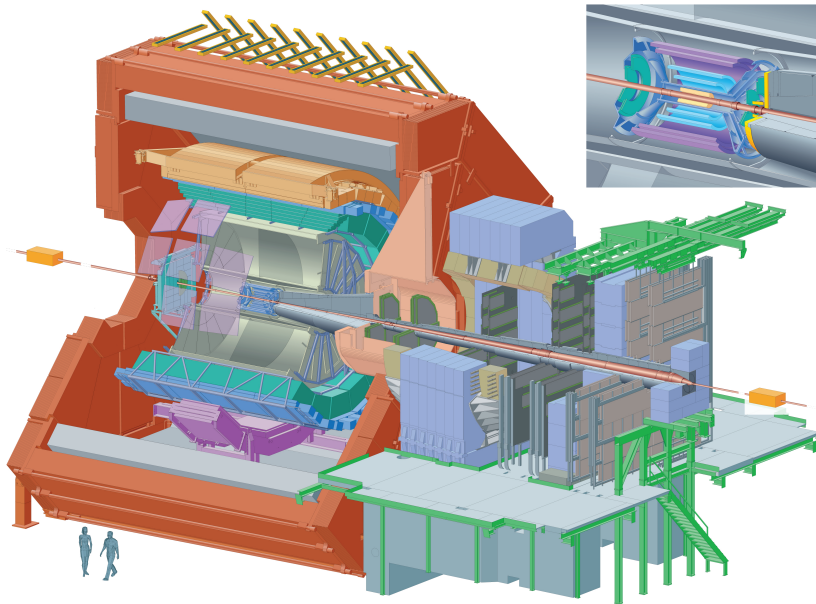


Figure 3.1.: Setup of the ALICE detector system [24].

The purpose of LHC is to provide the experiments along its circumference with high energy particle beams. Two beams are circulating inside the magnets of the 27 km long LHC ring and are collided inside experimental caverns about 100 meters underground.

3. A Large Ion Collider Experiment

Here, the collisions are recorded by the four main experiments - ALICE, ATLAS, CMS and LHCb - before being transferred to central storage systems to be studied.

3.1. Detectors

ALICE has a classic collider setup with layers of tracking (and vertex) detectors at the center closest to the interaction point and calorimeters in the outer layers [25], [26]. Triggering detectors are strategically placed according to their function, like T0 and V0 which both have detector elements placed on either sides of the interaction point along the beam direction.

The central barrel consists of a Time Projection Chamber (TPC) and Inner Tracking System (ITS). The TPC is the largest contributor of data in the ALICE experiment and the main tracking detector. The two end-plates of the TPC each have 18 sectors, and each sector contains an inner and an outer readout chamber. These are multi-wire proportional chambers with cathode pad readout [27]. In total there are 560000 pads in the TPC and they are readout by Front-End Cards (FECs) that each can handle 128 pads - or channels. This means that each sector needs 121 FECs. A schematic view of the TPC is shown in figure 3.2.

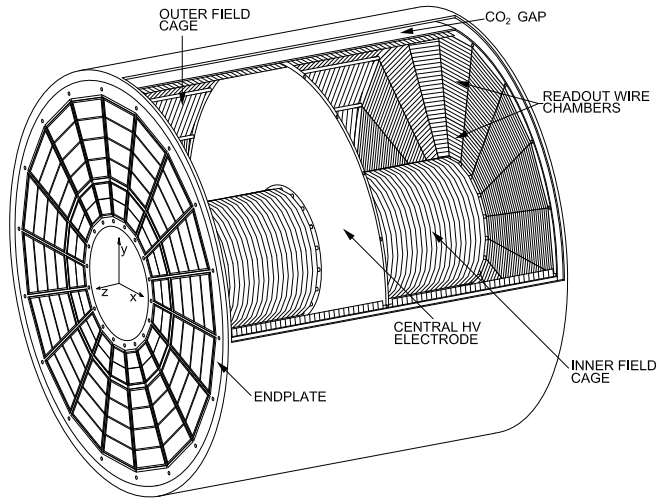


Figure 3.2.: 3D view of the TPC field cage [28].

For the first run, the FECs were organized in 6 rows, each row being connected to a Readout Controller Unit (RCU) with an attached Detector Data Link (DDL), as shown in the overview picture in figure 3.3.

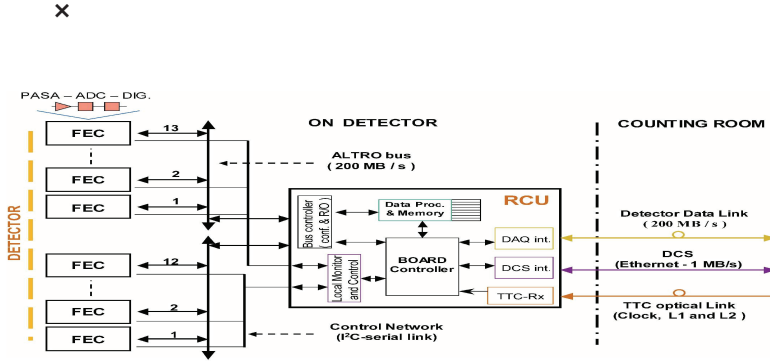


Figure 3.3.: Schematic view of front end and readout electronics [28].

3.2. Trigger

The Central Trigger Processor (CTP) combines information from all the triggering detectors and sends out trigger signals to Local Trigger Units (LTUs), where the final signals are prepared and forwarded to (read-out) detectors. The entire trigger sequence in ALICE is synchronized to the LHC clock. This signal along with the orbit signal (bunch crossing) is received and distributed by modules in the Timing, Trigger and Control system (TTC) in close coordination with the CTP. They are important for defining the time-window for when a collision can be expected to happen. Following the crossing of the bunches, triggering happens in quick succession through several predefined stages or levels [22].

The high multiplicities during Pb-Pb has made it necessary to use electronics with varying trigger timing needs for the different detectors. The typical “fast” trigger has therefore been split in two; L0 and L1, at $\sim 1.2\mu\text{s}$ and $6.5\mu\text{s}$ respectively. It is for the same reasons of high multiplicity that central collisions must be sand-boxed in their own event by past-future protection to be reconstructible. This process corresponds to the final hardware-trigger, L2, that lasts for $88\mu\text{s}$, which is also long enough to run (hardware) trigger algorithms [22].

After a trigger sequence has completed, read-out is initiated and the detectors are busy while extracting information from the event. The data is forwarded to DAQ/HLT systems where the final decision is made regarding the destiny of an event.

3.3. Data Acquisition

The data flow in ALICE starts when a L2 accept is issued. Detector signals are amplified, shaped and digitized in the front-end electronics and transferred to the DAQ and HLT systems via optical fibers.

DAQ receives the data in Local Data Concentrators (LDC), that can host up to two DAQ - Read-Out Receiver Card (D-RORC) cards. Here event fragments are combined into sub-events. These sub-events are sent to Global Data Collectors, where the complete event is built and forwarded to temporary storage where the data is recorded while the run is ongoing. After a completed run, the data is shipped to CERN Advanced STORage manager [29] (CASTOR), where it is made available for analysis on a large computing Grid [27].

In the D-RORC of the DAQ system, an exact copy of the data stream is made with the help of optical splitters and forwarded through 454 DDL links to the HLT. After passing through the HLT processing chain, the output is sent back to DAQ through 12 fiber links [27].

A schematic view of this data flow can be seen in the two figures 3.4 and 3.5. The LDC and GDC machines of DAQ are commodity hardware, much like the nodes in HLT.

3. A Large Ion Collider Experiment

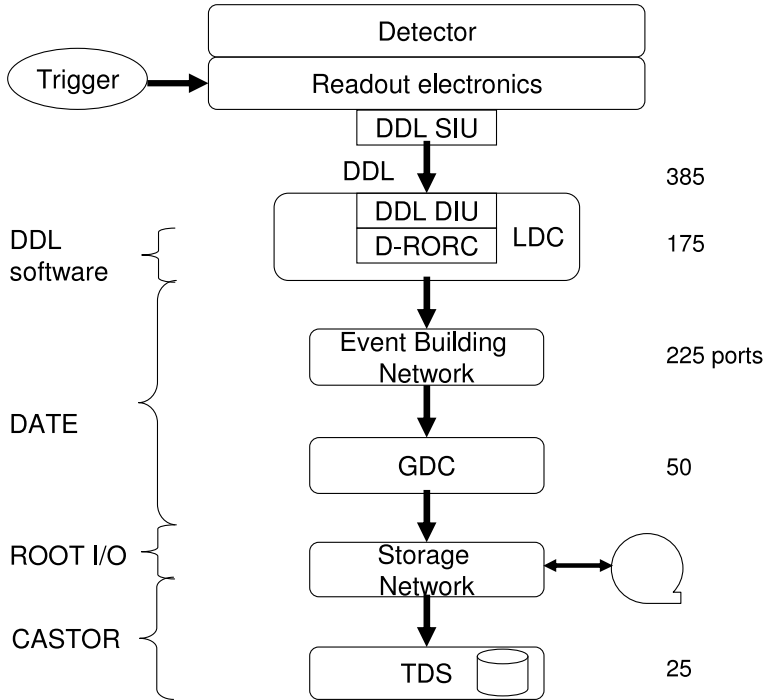


Figure 3.4.: Overview of the data flow in ALICE, from readout electronics to when being made available for analysis on the Grid. TDS is the temporary data storage where data is buffered until the end of a run, when its finally published to CASTOR [22].

3. A Large Ion Collider Experiment

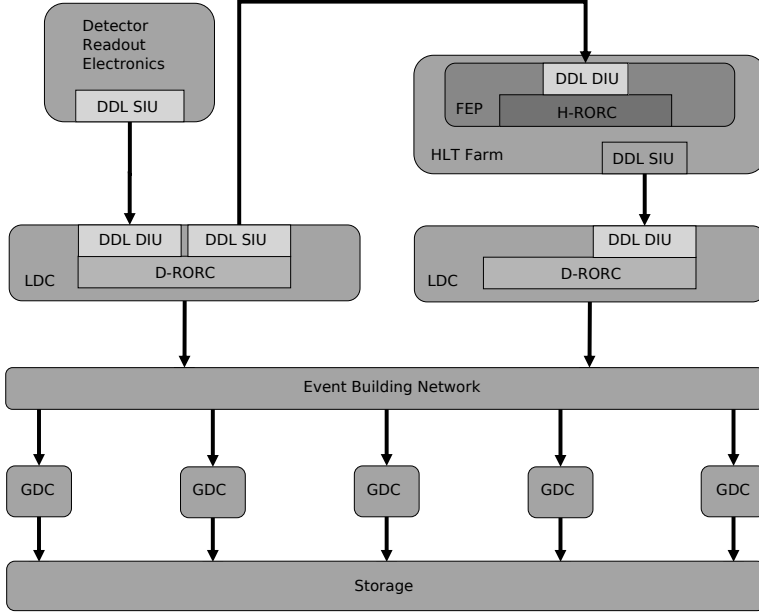


Figure 3.5.: Overview of the logical DAQ components and the interaction with HLT[22].

3.4. Control systems

The ALICE experiment is operated and monitored by the Experiment Control System (ECS) and the Detector Control System (DCS). These systems allow the entire experiment to be controlled from the ALICE control room by trained shifters. While training is essential for efficient day-to-day operation, it is still very important that the systems are designed in such a way that the shifter doesn't need to have intimate knowledge about each and every sub-detector, and all the support systems of ALICE, to perform their tasks. Much effort has therefore been put into the user interfacing parts of the control systems, to only present essential information and only expose relevant controls there.

The primary task of the DCS is to ensure safe and correct operation of the experiment, while ECS is the top-level control instance sitting atop of DCS and the other so-called "online systems"; trigger, HLT and DAQ [27]. DCS is in this sense more directed towards low-level instrumentation, allowing remote configuration, monitoring and control of experiment equipment, while ECS is more concerned with higher-level operations.

With a long commissioning phase, it has been important to design both these system to allow for partitioning of the experiment into smaller sub-systems in order to have efficient testing. With partitioning, testing can be done by the individual sub-detectors independent of the rest of the experiment, meaning it can for the most part be done

3. A Large Ion Collider Experiment

in parallel. This partitioning is possible due the distributed and hierarchical decision making of a Finite-State Machine (FSM) layer [30] that have been introduced in the Supervisory Control and Data Acquisition (SCADA) system used by ALICE and the other LHC experiments [27]. This system, called Joint Controls Projects [31] (JCOP), is a framework based on the commercial Prozess Visualisierung und Steuerungs System [32] (PVSS) package with additional components added according to the needs of the experiments.

The measurements and parameters collected by DCS are also needed for the calibration of the offline system when analyzing event data. Calibration data is collected from the PVSS database and detector algorithms - running on DAQ machines - to be made available for the analysis in a dedicated Offline Conditions Database (ocdb).

DCS also takes care of safety by interfacing all sensory systems in the experiment cave that has to do with the operating environment of the experiment, like gas, smoke, electricity, water and so on. One important design goal is therefore to allow for easy integration of a broad range of equipment, from power supplies and VME crates to computing devices and detector-specific electronics.

3.5. Offline Grid analysis

The Offline Project has developed the software for simulation, reconstruction, calibration, alignment, visualization and analysis. All the original data from the experiments is stored at two Tier-0 computer centers, one located at CERN and another in Hungary. The responsibility of safe storage of this data is shared with large regional computing centers called Tier-1. It is also these two infrastructures that share the majority of the organized data processing load. Tier-2 data centers use Tier-1 for safe storage and do not have permanent storage themselves. These are typically used for simulations and end-user analysis [27]. This hierarchical organization in tiers is inspired by the Monarc model [33].

A Grid middle-ware called AliEn [34] was developed by ALICE to orchestrate the distribution and processing of data, and MonALISA is used for distributed, aggregated monitoring [35].

After following the information flow from collision through detection, digitizing, triggering and data acquisition to permanent storage, the data is finally analyzed on the Grid infrastructure, producing physics results that are then published to the wider scientific community.

4. ALICE High Level Trigger

This chapter starts with a general introduction to triggering in HEP, before looking at the requirements for HLT in ALICE and the conceptual design that defines the final architecture. The remaining sections describes, in some detail, an HLT that is operational today and that have come a long way in reaching the functional goals it was designed for.

4.1. Triggering in High Energy Physics

Accelerators in High Energy Physics are typically linear or circular, although both types can be used together in the same accelerator complex at different stages, as is the case in LHC. Collisions happens either at a fixed target or by having two beams of particles collide head-on with each other. The collisions themselves are what creates interesting observables and as such they are called events.

At its core, HEP experiments are looking for signals in the event data that signifies information that could be interesting physics. Much of the data produced by the LHC experiments is in this regard considered uninteresting, as it contains already known physics and can therefore be discarded to save storage space and processing time [36].

The purpose of triggering in HEP experiments is to inform acquisition systems (DAQ) about interesting events, so that data can be retrieved from detector electronics and saved to permanent storage. Also within the event itself, there can be data that does not contribute to any new physics. Selection of the regions that are interesting (Region of Interest (ROI)) is therefore often provided for in triggering systems, as is also data compression to further reduce the size.

A typical experiment setup in HEP is therefore composed of, in addition to the readout detectors, also triggering detectors, that will initiate readout of data from Front-End Electronics according to predefined criteria. This is done in several stages, each considering larger parts of the events, but less data in total, as many of the events have already been discarded. Each successive step can therefore spend a little longer time, using increasingly more elaborate rules, in deciding which events should be passed on further.

When events happen, it will in general take some time for the processes that creates readable signals to complete. For instance, in drift chambers, there will be a delay from when the particles pass through the gaseous volume until the induced charges, representing the trajectory, reaches the electronics where it becomes available as signals to trigger systems. This is the trigger latency of the system.

There is also readout latency, the time it takes to read out data, that might differ amongst the detectors due to differences in electronics. A challenging part of building an

4. ALICE High Level Trigger

experiment is to align all the participating systems in time, so that it becomes possible to produce an accurate representation of the event.

While trigger electronics are designed to process events continuously, the detector read-out rates only need to match the expected average rate (including the readout latency) of a particular trigger condition. If the event rate outpaces the rate at which readout electronics can extract data, a situation arises where the detector is not able to accept events. This state of being busy shipping data is called dead-time [36].

Usually there will be hardware levels and a software level. The software level is what is normally referred to as a High Level Trigger. This type of trigger reconstructs the event data online and uses elaborate software algorithms to decide whether or not to keep an event. Being software based, and having the time to reconstruct and analyze the events, makes it a lot more flexible and powerful than its hardware-based siblings. Complex and efficient event filtering is becoming more and more important with the increasing trigger rates and data volumes of today [36]. Finally, a detector can also be constructed to self-trigger, by continuously reading data from front-end electronics and performing online (partial) event reconstruction that can be used to evaluate if the given event should be selected or not [36].

In a software trigger, while the trigger sequence is ongoing, the data that is potentially to be stored to disk - pending a trigger decision - must be kept in a pipeline. Also here, if they arrive faster than they are processed, the events will start to pile up in a buffer until there is no more space left, at which point the sending system will have to be told to stop sending events until space has been freed up in the buffer.

4.2. Requirements

In the ALICE HLT, the amount of data coming out of the experiment after hardware triggering, can still be as high as 25 GByte/s. The storage bandwidth, however, is restricted to about 4 GByte/s and it is therefore necessary to introduce a HLT that can reduce the data stream to permanent storage accordingly.

The ALICE HLT has been designed for the event selection and data compression needs of ALICE. It is a compute cluster consisting of about 200 high-performance compute nodes each made out of commodity computer hardware. Internally in the cluster, event reconstruction is performed online before the compressed events are propagated to event selection and triggering. The triggering decisions and ROI information can then be used by Data Acquisition systems to select interesting events in a collision [22].

From a physicist point of view, HLT should function as just another tool or instrument. The successful implementation of a HLT has to strike a balance between providing a powerful tool for online data processing and practical/economical implementation from a computer science/maintenance point of view. The high complexity of the system combined with strict requirements is what makes the construction of HLT challenging.

4.2.1. Functional requirements

The over-all objective of HLT is online data reduction without loss of quality. The main functional requirements are given by the chosen strategies of achieving that objective [27]:

- Event selection. (Trigger: accept or reject events based on detailed online analysis).
- ROI selection. (Select: Selects a physics region of interest within the event by performing only a partial readout).
- Compression (Compress: Reduce the event size without loss of physics information by applying compression algorithms on the accepted and selected data).

In general, the steps to accomplish this is to first read the data and reconstruct space points, also called clusters. From the clusters, tracks can be reconstructed and from these the global events are built. The reconstructed events are fed into the online analysis, which output is used in the trigger algorithms that finally send the trigger decisions to DAQ.

A trigger decision contains a list of DDLs to be read-out, where all links are included in the case of an accepted event and none for a ejected event. ROI selection is therefore in this context effectively a selection of a set of DDLs.

4.2.2. Non-functional requirements

The online aspect of HLT is an important difference to typical compute clusters; data reduction is performed as the data stream travels through the system without any intermediate persistent storage. Normally, the challenge in HPC is to run stable long enough for the jobs to be able to finish. A modest fault rate in a batch system is not a serious issue as the calculations can be restarted if something goes wrong. In contrast to this, any faults, crashes or the like in an online system, results in less data taking time, which leads to less statistics, and this has a direct impact on the physics results.

In batch processing, the problem to be solved is known to a central, uniform entity, which does the job scheduling. The scheduler divides and distributes the problem to worker nodes and gathers the results when jobs have completed. Whereas in HLT, the structure of the data set is intrinsically given by the geometry of the experiment and the units of work are the estimated - by benchmarking - loads of the individual analysis processes.

The geometry along with the read-out infrastructure (in the way data arrives on the DDL links) of the experiment therefore greatly affects the design and the layout of the ALICE HLT cluster. Taking the TPC detector as an example; with each HLT node being equipped with four DDL inputs, all four links from the outermost readout chamber of one sector are mapped to a single node. For the inner readout chambers, two pairs of links, each pair originating from a sector, are attached to one node.

In each rack there are three FEP nodes, two for the outermost readout chambers and one for the innermost chambers, organized so that each rack covers two sectors. Such

4. ALICE High Level Trigger

an organization ensures that at most two inner sectors or one outer sector is affected if a node fails, and only two sectors are affected if the entire rack fails.

From the read-out electronics, the data arrives at the HLT in fibers, with a granularity and composition that immediately allows for analysis components to start processing it independently from neighboring information. This means that the processing in HLT can be highly parallel. For instance cluster finding in one padrow of the TPC is independent of cluster finding in the padrow next to it. This is similar for most detectors included in HLT.

The stages of processing are also easily separable. This is maybe most clearly seen in the TPC processing, where first, clusters are found in the raw data, then tracks are found based on the cluster information. Tracks are first calculated within a limited region, like a pad or sector in the TPC, then these tracks are merged with matching tracks considering the combined detector volume. From this information, a global event is constructed, that is fed to the event selection. In the end, data compression is done. This processing hierarchy is illustrated in figure 4.1.

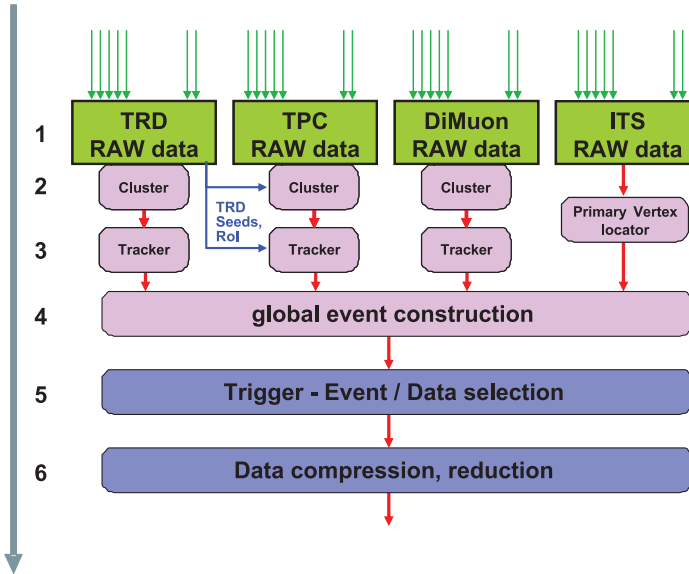


Figure 4.1.: The six architectural layers of the HLT [27].

Clusters of today are built with COTS hardware and software, which promises lower cost and shorter implementation time compared to custom solutions [1]. The drawback of off-the-shelf products is that the feature set of a product might not completely match the requirements of the customer. In particular, strict requirements regarding quality, reliability and real-time performance may not be met in a system using off-the-shelf

4. ALICE High Level Trigger

components.

The inherent risks of faults in COTS-based HPC clusters are further amplified in HLT due to its online nature, its complex software configuration, and the strict requirements for stability and reliability (stability in the sense of sustained operation over time and reliability in the sense that the produced information can be trusted to be correct). This makes it hard to ensure uninterrupted operation during a run. To meet this challenge, HLT will have to be made resilient to potential failures both in hardware and software by building in fault-tolerance.

Improved processing power is the reason clusters have become a successful architecture for supercomputers, but clusters are a general approach to improve one or more aspects of computing. Several aspects can be addressed at the same time, so that one can have both high performance and improved fault-tolerance, depending on the chosen design.

The current state of research of fault tolerance in very large systems is discussed in [37], which also features a detailed presentation of classes of fault tolerance and outlines the motivation for future research in the field. The presented strategies are mainly those of risk management; try avoiding faults to happen in the first place, try to minimize the effects of faults and repair faults when they happen. The type of fault tolerance sought built into the transport framework is active replication, which belongs in the “failure effects avoidance” class.

Choosing COTS also makes it possible to get quickly started prototyping on small systems that can gradually be expanded by adding more nodes as development progresses. Over time, software can be moved from one hardware generation to the next with little to no effort as long as the CPU architecture remains similar. The challenge is making sure that the software scales well with the number of nodes.

A recurring strategy is therefore to buy hardware late or on-demand, getting the most out of Moore’s law cost/performance-wise. This has also been the strategy for HLT, which has been expanded according to the progress and stages of the experiment. For instance when new detectors have been introduced in the processing chain or when higher data rates from the detectors - i.e. due to higher energies or higher luminosity - increase the demand for compute power. The computing needs are evaluated ahead of such events and based on estimates, new equipment is acquired.

4.3. Hardware

Hardware-wise, the main components of a cluster are the nodes - the processing units, interconnect, storage and input/output. Specialized hardware can also be used to improve the efficiency of cluster management and thereby reduce operational cost. In ALICE, the on-site computing facilities are organized in four rooms close to the pit, as can be seen in figure 4.2. The level marked as CR2 is dedicated for the production environment of HLT, while the development cluster is sharing CR3 with DCS.

4. ALICE High Level Trigger

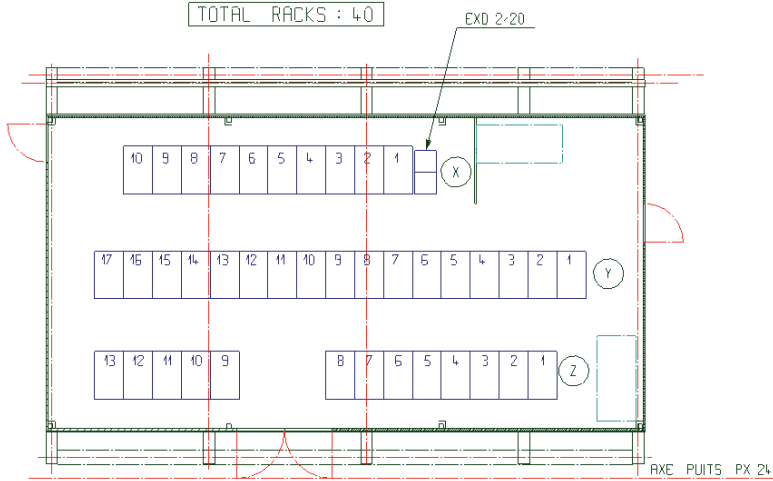


Figure 4.3.: The HLT counting room CR2 [38].

The following will be a brief description of the hardware in HLT. For a more detailed treatment of the subject, see [39].

4.3.1. Cluster nodes

A compute cluster is typically homogeneous in the sense that all the compute nodes are composed of similar hardware. This makes maintenance and resource management easier. In applications that process data in isolated steps, each with its own characteristics, overall performance can be increased by employing a heterogeneous cluster, where each machine type is tailored to a specific step in the processing chain. In HLT, there are two types of cluster nodes: FEP nodes and compute nodes.

FEP nodes are the machines where the optical fibers from the experiment enter HLT. These nodes are equipped with DMA engines that transfer the data stream directly into the memory of the host.

There are two generations of compute nodes. The earliest being equipped with a large number of disks and being enclosed in traditional rack chassis. In the later generation, the node unit is a blade that is inserted into a chassis that can hold four blades at the time. The blades can also contain a GPU instead of a motherboard, where the GPU is connected to the motherboard of the blade below, meaning that there are two types of the latest generation of compute nodes:

CPU node: General purpose compute nodes that can be organized in any way that makes the most sense for a given configuration.

4. ALICE High Level Trigger

GPU node: Nodes that have a powerful GPU, in addition to the CPU, that increases the performance for certain types of processing. The GPUs used in HLT are NVIDIA GTX 480 and NVIDIA GTX 580, having respectively 448 and 512 computing units each.

Infrastructure and portal nodes are mostly similar to compute nodes in terms of hardware, but typically also have extra equipment required for their task, such as Redundant Array of Inexpensive Disks (RAID) controllers for storage servers.

The gateways, used for outside connections to the cluster, are smaller dedicated systems that are less demanding when it comes to hardware. 1u systems with the equivalent processing power of a light laptop manage these tasks fine as long as they have enough Ethernet ports to act as a gateway and firewall.

In total 117 FEP nodes are installed, 84 compute nodes - of which 64 are equipped with a GPU, 4 portal nodes and 20 infrastructure nodes. The accumulated CPU core count is 2740 and the total amount of memory 5.29 TB for the processing nodes [40].

4.3.2. Input and output (H-RORC)

Detector data transport is accommodated by fiber technology (DDL) and data formats that are developed specifically for use in ALICE [22, 41].

The data stream coming from the experiment via DAQ enters the HLT through H-RORCs (see 4.4). These are PCI cards with a FPGA co-processor that pumps the data traffic into the HLT farm. A node can host at most two H-RORCs, each equipped with two Destination Interface Units (DIUs) daughter-boards. The DDLs connected to the DIU/Source Interface Unit (SIU) have a maximum bandwidth of 160 MB/s, while the bandwidth over the Peripheral Component Interconnect (PCI) bus is 370 MB/s for each H-RORC [27].

The H-RORC uses Direct Memory Access (DMA) to transfer the incoming data stream from the on-board optical receiver (DIU) (via decoding and digitizing) directly to the memory of the host without spending Central Processing Unit (CPU) cycles in the process. The same card performs the reverse process pushing data to DAQ via the output links of HLT, but in this case using SIUs, the counterpart of the DIU [27].



Figure 4.4.: H-RORC data pump and FPGA coprocessor [27].

4.3.3. Processing

The processing characteristics of the target application will typically be the most important factor in determining the composition of computing devices. Some clusters will make heavy use of GPUs while others will only use CPUs.

The logical separation of processing steps in HLT, as seen in figure 4.1 and the characteristics of the involved algorithms, opens up opportunities for enhancing the performance with the help of coprocessors in a heterogeneous processing environment. Such co-processors have therefore been important components in the HLT design from the very start.

The first step of the reconstruction process, cluster finding, can be done in the hardware logic of the FPGA co-processor on the H-RORC [42]. Tracking, the second step, can be done in GPUs [43]. The analysis components for both these steps were first developed as software for easier verification of the algorithms and then the hardware version was later extensively tested and verified against its software equivalent [43]. Both these co-processors were successfully commissioned in time for heavy-ion runs in 2011, where the increase in performance from these additions were crucial for achieving the required data rates [44].

4.3.4. Storage

Another important design principle of HLT is to always keep the data that is being processed in memory from it enters the Front End Processor (FEP) nodes until it leaves HLT. This way, there will be no slowdown of the processing rate due to disk operations. Compared to a traditional batch cluster with a shared data set, the distributed nature

4. ALICE High Level Trigger

of the input and the hierarchical processing chain in HLT, makes it easier to implement such a design. Nevertheless, for distributing information - such as when configuring the application - to the compute nodes, a shared storage is needed also in HLT. This storage needs to be highly efficient with regards to concurrent access and bandwidth so as to keep the setup time for preparing the application low.

4.3.5. Interconnect

Clusters relies on network to move data between storage and processes. Low latency and high bandwidth are increasingly more important as higher performance is sought. There are three separate networks in HLT:

- The physics application in HLT makes use of Infiniband to achieve high enough rates for the data flow.
- All other traffic, like logging and most management tasks, use Gb Ethernet.
- An out-of-band management network (100 Mbit) that connects to either CHARM cards or IPMI/BMC-enabled ports on the nodes are used for lower level management and remote diagnostics.



Figure 4.5.: Picture of counting room 2 at the ALICE experiment where the HLT is located. In the racks, FEP machines are the ones in the middle and compute nodes are in the top where also the fibers from the experiment can be seen. Switches and infrastructure nodes are placed in 4 racks to the left at the very far end of the two rows seen in the picture.

4.3.6. Cluster management (CHARM card)

To reduce operating cost, extensive remote management solutions were included in the plans from the very start. The original design of the management solution included functionality for [45, 46]:

- Automatic installation and self-test of nodes.
- Simple automatic diagnostics and recovery of crashed machines.
- Extensive monitoring of temperature inside the chassis of the nodes.

This was to be implemented by using a dedicated management card, the CHARM card, and management software that later became SysMES.

The CHARM card is a PCI card with a FPGA, running a small linux system. The card looks like a normal graphics card to the host and can forward the video signal over the

network as a Virtual Network Computing [47] (VNC) connection. In addition to remote view functionality, the card is also equipped with temperature sensors, connectors for controlling the power of the host and an interface for mounting installation disk images on the host over USB [48].

After Intelligent Platform Management Interface (IPMI)/Baseboard Management Controller (BMC) [49] became a wide-spread standard common in server solutions, the need for a custom management card diminished and the production was stopped. It is now only the oldest FEP nodes that still have CHARM cards and these will only be supported until the next generation of RORC cards become available, at which point the FEP nodes will be upgraded to nodes with IMPI functionality included.

4.4. Software

Apart from the design and composition of the cluster itself, the H-RORC and CHARM card, HLT is mostly a software project. Notable software contributions to the HEP community are:

- The development of a HLT analysis framework and processing components (for ALICE type of experiments) tuned for online use, with an emphasis on reuse of existing offline code [50].
- The development of a flexible data transport framework that handles the traffic between distributed components in a cluster [51].
- Integration of HLT analysis components into transport framework and the offline environment (AliRoot) [50].
- Ports of certain software analysis components to hardware for increased performance [43, 42].
- The development of a flexible triggering framework that can be used for event selection (briefly described in [39]).
- Development of compression components [52].
- Integration with the control systems at the experiment site [39].
- Implementation of data exchange protocols to relevant databases such as the OCDB [53].
- Development of Cluster monitoring and management software for High Performance Computing [54].

4. ALICE High Level Trigger

4.4.1. Software system anatomy

The source code repositories of the software used in the HLT project are listed in table 4.1. The `hlt-alice` repository contains transport framework, interfaces, hardware firmware and configuration tools for the HLT analysis chain. The `alroot` repository is mainly used for the development of offline code in ALICE, but also contains the online analysis components of HLT. It builds on the the Root data analysis framework [55] that is developed at CERN and used throughout the LHC experiments.

Project name and source location	Description
<i>hlt-alice (all)</i> svn+ssh://svn.cern.ch/repos/hlt-alice	Transport framework, interfaces, control
AliRoot (reposable for HLT directory) http://git.cern.ch/pubweb/AliRoot.git	Analysis components
<i>SysMES (all)</i> KIP internal subversion	Distributed monitoring
<i>infologger (small percentage)</i> hlt cluster	Application logging
root (none) http://root.cern.ch/git/root.git	Dependency of AliROOT
<i>node-config (all)</i> hlt cluster	Scripted node configuration
<i>documentation (all)</i> hlt cluster	Internal documentation

Table 4.1.: Repositories containing software used by HLT. The involvement of HLT is indicated in parenthesis next to the project name.

Further details of the sub-projects or modules can be seen in table 4.2.

4. ALICE High Level Trigger

project	critical	desc
alroot:HLT	True	analysis
hlt-alice:BCL	True	support lib
hlt-alice:Build	False	build/release
hlt-alice:Components	True	data routing
hlt-alice:EVTREPLAY	False	debug/monitor
hlt-alice:Framework	True	data routing
hlt-alice:H-RORC	True	hardware
hlt-alice:MLUC	True	support lib
hlt-alice:PSI	False	depricated
hlt-alice:PSI2	True	kernel/hardware
hlt-alice:RFLASH	False	hardware/utility
hlt-alice:RunManager	True	operation
hlt-alice:SVN-Release-Tools	False	build/release
hlt-alice:SimpleChainConfig1	True	configuration
hlt-alice:SimpleChainConfig2	True	configuration
hlt-alice:TPC-OnlineDisplay	False	debug/monitor
hlt-alice:TRD-Readout	False	debug/monitor
hlt-alice:TaskManager	True	operation
hlt-alice:Utility_Software	False	debug/monitor
hlt-alice:bigphys	True	kernel
hlt-alice:control	True	operation
hlt-alice:docs	False	documentation
hlt-alice:interfaces/DDLTestGui	True	interface
hlt-alice:interfaces/HCDBManager	True	interface
hlt-alice:interfaces/ecs-proxy	True	interface
hlt-alice:interfaces/fed-portal	True	interface
hlt-alice:interfaces/pendolino	True	interface
hlt-alice:interfaces/taxi	True	interface
hlt-alice:invdb	False	documentation
hlt-alice:monitoring	False	debug/monitoring
hlt-alice:restricted	False	admin
hlt-alice:tools	False	utility
hlt-alice:trunk	True	configuration
hlt-alice:util	False	utility
hlt-alice:utilities	False	debug/monitor

Table 4.2.: Software packages from the hlt-alice and alroot repositories with a short description. The packages necessary for HLT operation are those with critical set to True. Projects labels are composed of <reponame>:<projectpath>, where project path can be a sub-directory.

4.4.2. The physics application

Due to the nature of the data that enters the system, HLT has been designed with a distributed and hierarchical structure, where software components consider increasingly larger parts of an event. The application software developed for HLT consists of:

- A publisher-subscriber framework that provides a way for analysis processes running in parallel to talk to each other.
- The runtime management and control of these components and their communication is done by control software called TaskManager.
- The initial analysis chain setup is created by chain configuration tools based on input from templates, script-snippets and configuration files.

An overview of some of the software components and operations in HLT can be seen in 4.6.

4. ALICE High Level Trigger

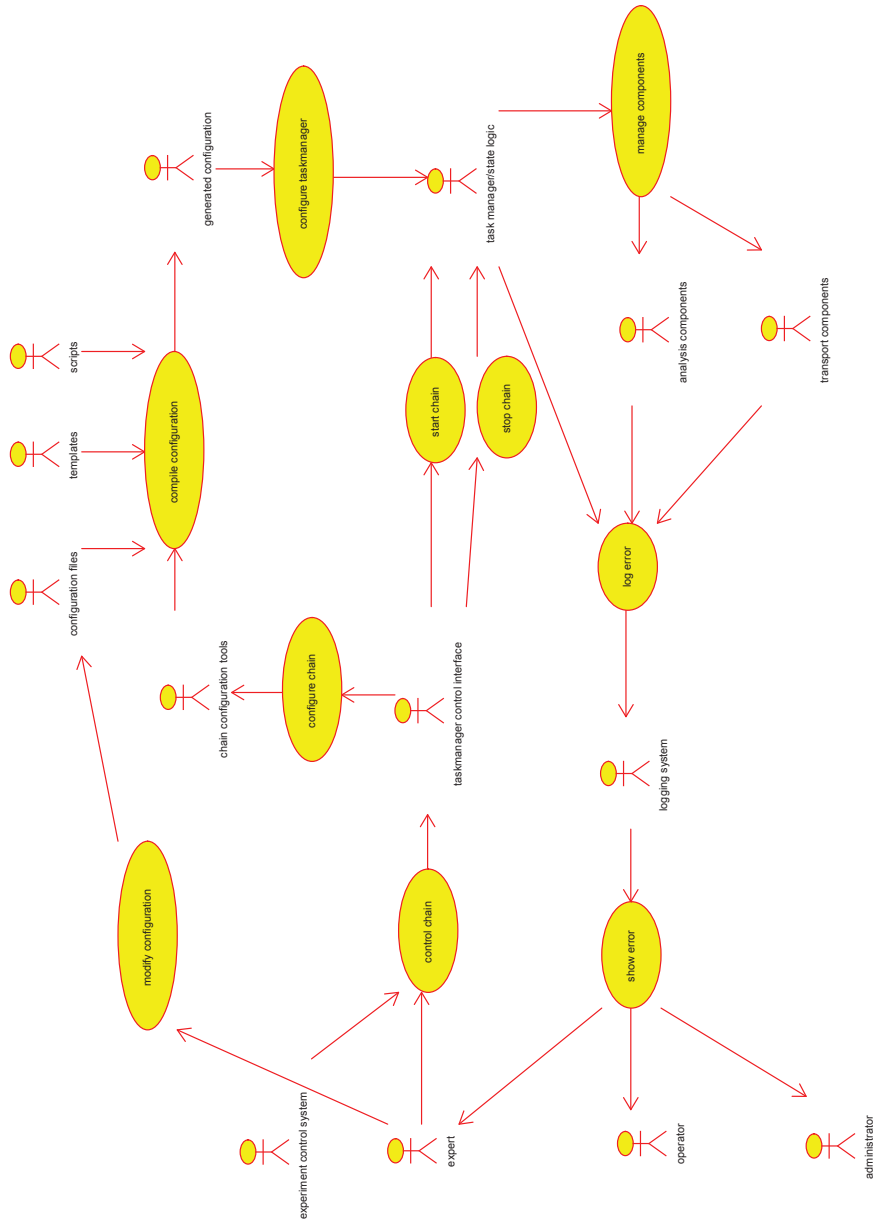


Figure 4.6.: Software components and actions involved in chain operation.

4.4.2.1. Analysis framework

The stages of processing in HLT is divided in analysis components. Each type of component performs a certain task for a certain part of the event. This is where the actual computation in HLT is being done. The size of the input can vary from event to event, so resource estimates of a process that depends on input, must be according to a worst case scenario. The HLT analysis framework has a modular and well-defined inheritance structure, and it is designed to work both online with HLT and offline with aliroot [50].

4.4.2.2. Data transport framework

The highly parallel processing in HLT makes the overall conceptual mapping of the data flow mostly straight-forward. But when considering available resources on each node and the details of the data transport framework itself, the full process placement quickly becomes non-trivial.

The main tasks of the data transport framework are to distribute, gather and synchronize data streams, making sure data flows as fluently and efficiently as possible between analysis processes. The most difficult task being to synchronize the streams when event fragments of the same event have different processing time. An example would be when a padrow in the TPC has more cluster hits than the rest of the padrows that are fed into the same tracker from the cluster finders. In this case, the events will have to be queued up in the next processing stage until all needed event fragments are available for the processing to begin.

The framework has also been designed to support fault-tolerance by using active replication and re-routing in the case of node failure. The data transport framework used in HLT is described in detail in [56].

4.4.2.3. Chain configuration software.

Chain configurations are defined in XML-files with a set of domain specific tags related to the detector layout of ALICE and the hardware resources in HLT. These domain specific tags abstract some of details of the chain setup so that for instance identical sections of a symmetric detector only have to be defined once. The chain configuration software then has some knowledge about the detector layout that it uses to generate the exact configuration (defines structure of detectors and how components are chained together) based on the definitions in the configuration files.

Configuration files defining the available resources in HLT are the second type of input to the configuration software. These are also XML-files, but simpler in structure as they only define the nodes and their hardware configuration along with a list of available DDL links. The last type of input are script templates or snippets for starting the involved processes. All the configuration files and snippets are stored in the hlt-alice repository so that it is possible to keep track of their history and to roll back in case of misconfiguration.

A predefined set of configurations, each tuned for a certain beam condition, are made available to ECS which chooses one based on the type of run to be started. To be able to

correctly setup a chain on only functional machines, the low-level configuration files have to be generated before each run and is then distributed to all participating machines.

4.4.2.4. Interfaces

Figure 4.7 shows a conceptual overview of the many interfaces in HLT to external systems. Those connections that are critical for HLT operation (ECS, DCS) are implemented with redundant setups, including hardware, software and the physical connection itself. Changing to a hot spare should happen without any intervention from operators [53].

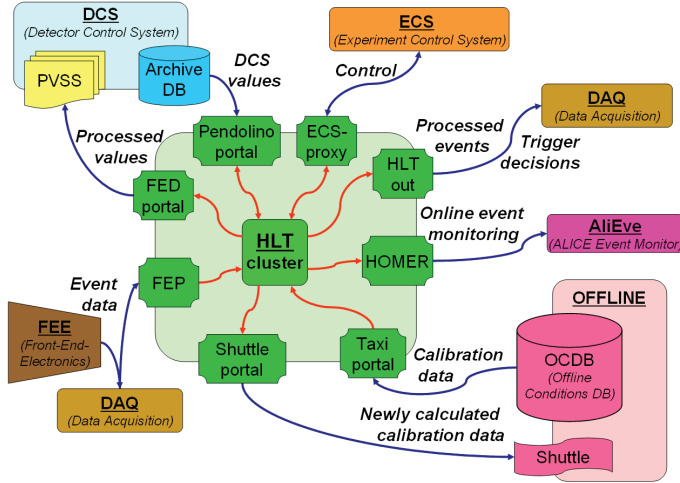


Figure 4.7.: HLT interfaces to other systems of ALICE [53].

The DDL fiber links [57, 58] are the data stream interfaces of HLT. The data packets exchanged over these links are encapsulated in a well-known format according to an agreed upon specification [59]. Conforming to this specification allows the readout chain in ALICE to establish appropriate protocols for ensuring that the correct data is produced.

The use of a FSM in the communication between control systems and sub-systems fits very well with the internal TaskManager hierarchy of HLT, having one master TaskManager as the single point of control for the analysis chain. Between this top-level process and the ECS, a proxy exists that maps the incoming state changes to corresponding internal commands.

Apart from data flow and control systems, external interfaces are mainly needed for calibration and monitoring:

- The Shuttle and Taxi portals, exports and imports, respectively, calibration data between the ocdb and HLT [60][53]. With this information, analysis components

4. ALICE High Level Trigger

can be properly calibrated before each run and the same system can recalibrate components while a run is ongoing if needed. Analysis components are also a source of calibration information in HLT. At the end of a run, all calibration objects are gathered by a special subscriber component in the processing chain and made available to be fetched by the offline shuttle [53].

- Additional parameters are fetched from the the DCS database, which contains information gathered from detectors and other equipment in the experimental setup. Pendolino is the software in charge of this task.
- Lastly HOMER is used for online event monitoring, whose one of its outputs, the event display, can be seen in the screenshot below 4.8. Details of the HOMER manager implementation in HLT can be found in [39].

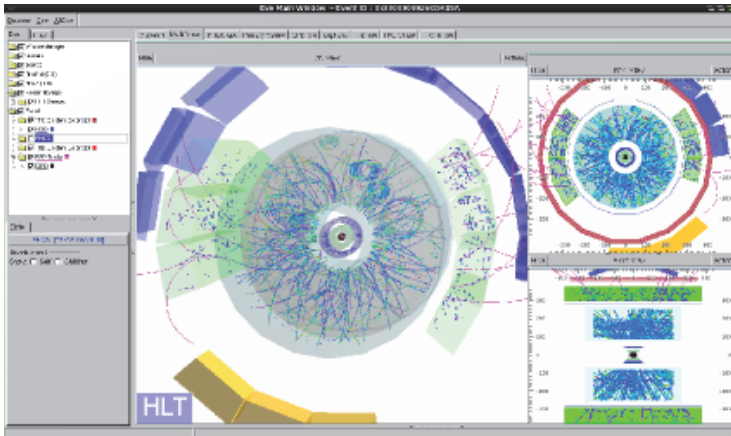


Figure 4.8.: The event display showing one of the first pp collisions at 7TeV [39].

4.4.3. Software environment

Like all software, also cluster applications needs a certain environment for performing its tasks. The most fundamental piece being the operating system, but the system services are also critical for the operation of the main application. Other components or modifications might also be needed depending on the requirements.

Initially, the Linux distribution Ubuntu was selected as the sole operating system of HLT because of its up-to-date packages and increasing popularity. Over the years, Ubuntu has become more and more of a desktop centered distribution and some of the latest developments creates problems rather than improve the situation for the use case of HLT. Other distributions and operating systems have therefore been put to use where these make more sense. For instance OpenBSD [61] for firewall, Gentoo [62] (host) for virtualization and Debian [63] (guest) for (virtualized) web services.

4. ALICE High Level Trigger

For the FEP nodes, a patch [64] was applied to the Linux kernel that allows - by configuration - the operating system to set aside large continuous blocks of system memory for use as an efficient way of importing large amounts of raw data and make it available to software running on the host system. This is where the RORC injects the data, with the help of the PSI [65] driver, so that it becomes available for the data transport and the analysis components.

The system services run on top of the operating system - and it is maybe the extent of needed infrastructure support that sets distributed architectures the most apart from others:

- Before being able to control a node, one must be able to address it. Identity services (DHCP/DNS) assigns ip-addresses and host names to all the nodes. With this in place, the nodes are able to communicate and other services can start.
- Access control for applications and users is normally implemented by a login (ssh) and a directory service (LDAP) that together perform authentication and authorization, and a distributed file system allows working on the same set of files from all nodes.
- Many services also rely on a common synchronized clock (ntp) for their operation, and such a common clock is critical for any meaningful centralized logging.

All these services need to be setup in a redundant configuration so that if a service or the hardware it runs on fails, there will still be at least one instance left for the clients to continue to work. This is typically achieved by having two or more entries in the client configuration each pointing to a self-sustained instance of a service.

Setting up such a system is a matter of configuring the services according to the needs of the project, but doing so requires a lot of knowledge and experience.

Lastly, in order to facilitate testing and in order to be able to quickly roll back in case problems should arise during operation, a mechanism is needed that allows for easy switching between application releases. In HLT, this is achieved by installing releases alongside each other on the same operating system installation and preparing scripts that change the active release with the help of a set of environment variables [39]. Environment Modules [66], is a commonly used tool for this purpose and is also used in HLT. It manipulates the system's environment variables to ensure the correct versions of libraries and binaries are loaded. A short description and case studies for this tool can be found in [67]. Wrapper-scripts using Environment Modules have been provided for most operator tasks. A thorough description of these scripts can be found in [39].

4.4.4. Cluster management software

Being a composed machine, a cluster requires elaborate software for supporting its operation. Examples of the many requirements that must be met include: to make efficient usage of resources, being able to react to failures in an effective way and making debugging easier. Several systems have been deployed in the HLT cluster to meet these requirements and will be briefly described in this section.

4.4.4.1. The SysMES Framework

System Management for Networked Embedded Systems and Clusters (SysMES) is an automated management solution for networked devices, primarily targeting distributed computing infrastructure like clusters. Clients running on a device communicates with a central server that can dynamically deploy monitor and action objects to clients from an operator layer. The client executes actions according to predefined rules and input from monitors, either autonomously or in concert with central servers [54].

HLT has been one of the reference installations for the SysMES framework and the SysMES team have been working in close collaboration with the rest of the HLT group to have their management solution deployed and working correctly on the cluster.

4.4.4.2. Infologger

Infologger is a centralized logging system that is part of the DATE [68] software package developed by the ALICE DAQ project. At the core is a central server that receives messages from infologger daemons running on all the client nodes. Messages are inserted into a MySQL database that have been setup (with a schema that matches the infologger message format). The overall architecture is shown in figure 4.9.

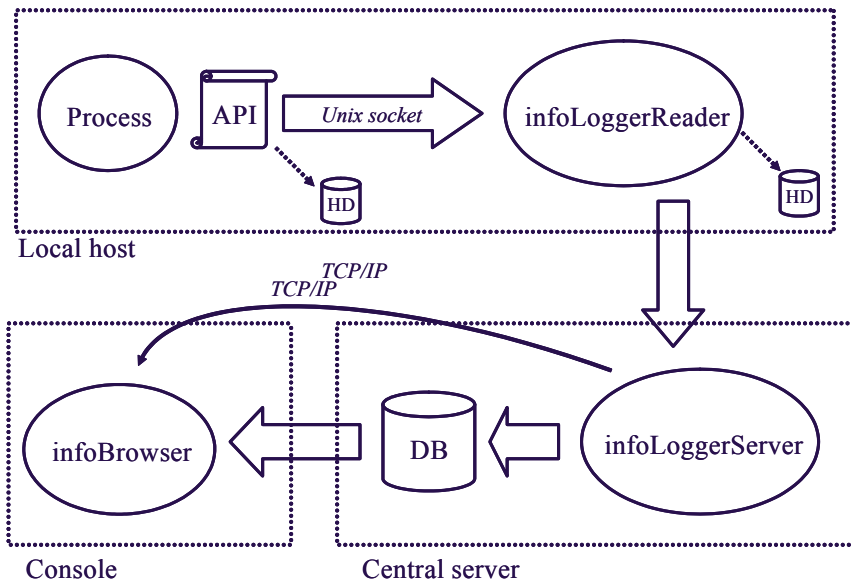


Figure 4.9.: DATE infoLogger architecture [69].

HLT software is linked during build time to a library that handles the communication

4. ALICE High Level Trigger

between the client program and the local infologger daemon. Finally, a GUI program, the InfoBrowser, shows the live stream of messages as they arrive at the server. This GUI can also be used to query the entire database for previously logged messages.

With small adjustments the infologger system was made to work with the HLT software. Messages from the analysis components are propagated through the analysis framework and the transport framework to the infologger system. The logs from the transport framework itself are also sent here.

The infologger installation for HLT was set up in a high availability cluster (Linux-HA) [70] with automatic failover from a primary node to a hot standby. The MySQL database was replicated with drbd [71] so that logs would be immediately available in the case of a failover.

4.4.4.3. Inventory database

Inventory databases are in general important for cluster administrators when managing a cluster. For instance to know which part to order when something breaks, to keep track of systems when reorganizing or to have relevant and up-to-date information available when debugging.

An inventory database was implemented earlier in the project as a typical lamp-stack [72]. Students were tasked with updating the database when new hardware arrived at CERN and then administrators and other personnel using the cluster were supposed to keep the database up to date during day-to-day operation.

The experiences from this first implementation was that such a database demands too much time to be kept up-to-date. Also, there will always be the risk of human error when information is entered into the system.

Part of the reason - at least an aggravating factor - was that there were a second place for node information to be entered; in the LDAP configuration that was also used for DHCP and DNS. At least IP address and node name would be required to be entered here for a node to be integrated in the cluster. This duplication meant two places to enter information that would have to be kept in sync manually.

This database has since been abandoned as the information became outdated and it was too much work to keep it in sync with ongoing changes.

5. Experience and performance

The large scope of a collider like LHC and the novel nature of any HEP experiment makes long term planning difficult during development, commissioning and first years of operation. Being part of such an environment, the internal planning becomes a daunting task for the participating collaborations, where schedules and requirements remain uncertain and can change. The complexity and ambitiousness of the ALICE HLT, makes it no less of a challenging environment.

This chapter therefore starts by giving a brief account of the initial phases, roughly divided in a hardware/installation section (5.1) and a software/development section (5.2). They serve as a backdrop for the performance evaluation that is the topic for the remainder of the chapter, starting with the more traditional and maybe simplistic ways, before turning to a more recent and comprehensive approach found in HPC productivity.

The chapter concludes with a summary of the major points discussed so far, with the intention to see what can be learned for similar projects in the future.

5.1. Installation, Commissioning and Upgrades

Due to a strategy of gradual expansion of the HLT cluster, several upgrades were planned and implemented as the LHC program unfolded. Alongside the planned activity come actions needed to tackle unforeseen issues. Solutions have to be found and the implementation has to be scheduled according to the LHC plans and available resources.

The general commissioning of HLT for first pp collisions is covered in great detail in [39]. Here, a chronological list of the most noteworthy events follows, to give an impression of how a gradual deployment can be carried out, and as a point of reference for later sections.

5.1.1. Node installation

The first infrastructure hardware and FEP nodes arrived in early 2006. By the end of summer there were enough machines installed to have test configurations running for a full TPC sector. These machines became the testbed for the early integration work needed to be done, and from this first small test setup, the cluster has been gradually expanded until it reached its production capacity for LHC startup in 2008 and beyond through the subsequent upgrades.

In 2007 the majority of the FEP nodes were installed, followed shortly by the first pure compute nodes in 2008. The FEP nodes were later upgraded with new CPUs and more memory. 117 FEP nodes are installed in total. During the preparations for the first heavy-ion runs in 2011, the cluster reaches its final size and target capacity with the

5. Experience and performance

conversion of 32 existing compute nodes to GPU nodes, totalling 84 compute nodes, 64 of them equipped with GPUs. In addition there are 4 portal nodes and 20 infrastructure nodes.

5.1.2. Network

The need for faster network connectivity was increasingly being investigated in 2010, until it was decided that improved bandwidth and in particular better latency would greatly benefit the HLT application. While Gb ethernet in a bonded configuration (using two physical ports as one logical port with the combined bandwidth of the two) might have been sufficient, a solution based on InfiniBand would leave enough headroom to be certain that the network would not become a bottleneck.

A rather large intervention was then undertaken, where InfiniBand was deployed on the cluster to replace Gb ethernet for the physics application. Cables were pulled throughout the counting room and switches installed. Old nodes were upgraded with InfiniBand cards, while all future nodes were bought with InfiniBand on-board.

The second major network intervention happened around the same time. Originally all Gb switches were placed in a central rack next to patch panels where all cables from the surrounding (CR2) racks were gathered. This layout led to a high cable density that was not helped by having all the cables of foreseen future nodes also installed early on in the project. Together with an intricate network configuration - i.e. some nodes with bonding and some without - it became a complicated network scheme, with cabling so dense and convoluted that it became a hindrance for efficient maintenance.

In early 2011 the Gb network cabling was therefore reworked and greatly simplified by distributing switches to racks closer to the connected machines.

5.1.3. Storage and system services

As the system grew after the first prototype installation in the counting room, Andrew File System (AFS) was set up as a distributed file system on the HLT cluster (ca. 2007). In combination with kerberos and ldap, it provides a complete single-sign-on, directory and file storage service for users and applications. This worked well until the cluster grew close to its final size, at which point problems started to appear that were not well understood. These issues were the main reason for replacing AFS by a more modern parallel file system during the winter shutdown 2010/2011 [40]. Currently, FraunhoferFS and glusterfs is deployed on the cluster. Each tailored for different needs; performance in HLT application and user homes respectively.

At the same time, LDAP/kerberos was removed completely from the cluster and replaced by normal user accounts and ssh with password-less login provided by public ssh keys.

5.1.4. Clustering and virtualization

Initially there were many dedicated machines for all the services and connections needed by HLT, such as HLT interfaces, infologger, core services and so on. Most of these are

5. Experience and performance

not demanding enough to warrant their own hardware, but such an approach is often the most straightforward initially and it satisfies the redundancy requirements in the case of HLT. Following the same pattern, the databases needed by services were installed alongside the application locally on the same machine.

Having many nodes and services that are similar, but still need to be treated independently results in inefficient use of resources and a rigid infrastructure that is unnecessarily hard to maintain, increasing the workload on the administrator. A scenario like this can be improved by centralizing services on a single powerful server, but this might come into conflict with other requirements, as in the case of HLT, where a single machine would not be able to provide the needed redundancy or fail-over capabilities.

Compared to a simple hot spare strategy, equally good or usually better fault-tolerance can be achieved by having clustering implemented at the application layer. In particular database and storage can benefit from having clustering built-in this way, also for improving performance.

At the hardware layer, one can further increase fault-tolerance and availability by sharing highly redundant storage over the network or high bandwidth storage connectivity (i.e. SAN). Other resources, like CPU and memory can be shared between services, either in the traditional way, by running on the same node and under the same OS, but also in a more flexible way with virtualization.

During the last few years, the prospect of virtualization of services has become more attractive in HLT and consensus became that it would be advisable to move towards consolidating hardware resources and services with the help of virtualization. With virtualization, the resources of a host can be partitioned in any arbitrary number of ways, so it's easy to make OS instances with resources matching the requirements of a service. In order to retain redundancy, the virtualization layer can also be clustered.

Clustering and virtualization has made it possible to create a flexible platform to deploy virtual machines for most of our services. The increased use of virtualization has led to easier management and more efficient use of resources as well as quicker recovery from failure. Instead of having 4-6 nodes each running mysql, there are now two in a clustered setup, sharing the same configuration. Instead of having many dedicated machines, there is now a general purpose virtualization cluster which can host virtual machines for services in HLT. There are fewer potential sources of error, fewer points of entry from a security point of view and a more unified way to manage services.

This has left mostly gateways as the only dedicated infrastructure nodes in HLT. But even here, the two entry points of HLT shares firewall configuration dynamically between redundant pairs of machines.

5.1.5. Big design up front

It is common practice to take advantage of Moore's law when designing a cluster and buy hardware as late as possible in the process to have the highest return on investment. But care must be taken in an industry where infrastructure and deployment can be influenced by rapid developments in its technology sector; when a cluster deployment is planned over many years and technology can substantially change within few years, there will be

5. Experience and performance

a risk that plans become outdated before the time arrives for installation.

Early HLT design documents spoke about a cluster of up to 500 nodes and 1000 CPUs [22], while the final installation counts 225 nodes (which should equate to 450 CPUs considering all nodes have dual-cpu motherboards) with a total of 2740 cores [40]. Plans had been made out assuming that the racks in the counting rooms would be filled from top to bottom. A scheme based on these assumptions was worked out for network layout and cabling, and IP addresses were calculated based on the placement of nodes according to this scheme.

Each time the compute farm was expanded, the computing density of the machines available on the market had increased. First higher number of cores in each box, then several nodes in the same space that earlier had housed one, then nodes were coupled with GPUs, each with a large number of computing units (448 for NVIDIA GTX 480 and 512 for NVIDIA GTX 580). The most significant change in HLT, the increase in computing density, has thus resulted in a much higher number of cores per network connection per unit height than what was planned for. And while the initial assumptions greatly influenced the deployment from the very beginning, the cluster layout was not evolved to reflect the later change of reality.

Instead of having racks filled with nodes, the computing density has increased to the point that only 4-7 boxes are needed in each rack to fulfill the computing needs of HLT (2012). In the end, the original schemes (section 5.1.2) were much reworked and simplified, but in the meantime, much time and effort was spent by administrators and others on-site struggling with a complicated and rigid installation.

These observations resemble the signs of “big design up front”, where detailed planning is performed early in the project, but the design documents are later not updated to reflect a better understanding of how the system should be implemented [73][74]. A strategy for reducing the effect of “big design up front” is iterative development which is discussed in section 6.5.2.

5.2. Implementation and development

Preparing for participation in a HEP experiment is a long journey spanning many years. It starts with simulations that are used to work out how the machinery should be put together. When the collider infrastructure is defined and the most important properties of the experiments are known, then the work of designing the data processing infrastructure can begin. During this phase prototypes are built, tested and evaluated. The output of these efforts are Technical Design Reports. These documents describe the systems to be built in sufficient detail for development to be started and the project to enter the implementation phase.

5.2.1. Developer infrastructure

A smooth and successful implementation phase depends heavily on a solid, common infrastructure. Shared source code repositories and prototyping environments are the

5. Experience and performance

foundations of a collaborative and productive environment in this respect. Even more so in an international project where institutes from all over the world participate.

Local prototyping clusters are used for testing smaller pieces in isolation, but for larger tests and for integration with other ALICE systems (control (ECS/DCS) and data acquisition (DAQ)), one will need a system available at the site early on in the preparations. The cluster, while being built in the “CR2” counting room of the ALICE pit, was initially used for this. But as commissioning grew closer, a second, smaller cluster was prepared in the same facilities so that development and testing could continue in a realistic environment without interfering with the main cluster.

A shared source code repository for HLT software - except for analysis components, which were already in CERN/ALICE svn repositories - was setup at CERN in early 2011, relatively late in the project. Before that, repositories were hosted by the respective participating universities/institutes, and required login credentials at the site for write access. In such a scenario, the threshold for contributions from other sites becomes higher than it needs to be and effectively prevents/hinders certain useful practices, such as “collective ownership of code” (section 6.5.6) in a distributed collaboration.

5.2.2. Debugging and testing

Running full-scale tests and trying to debug the system as a whole, causes the lack of good tools for large, distributed systems to become apparent. HLT being an online/real-time system further complicates matters, as the amount of information needed to reproduce a faulty condition, is so resource-demanding that extracting it would be very difficult to do without affecting the running system by using resources meant for the processing chain. One could add dedicated infrastructure for the sole purpose of debugging, essentially replicating large parts of the cluster, but that would be costly and probably difficult to justify.

There are systems that individually supports an aspect of the required tooling - like centralized logging, system visualization, data tracing/accounting - but the main challenge is to correlate and present the information in a helpful way to the operator and experts. Potential useful tools are discussed in section 6.6.

Difficulty in testing smaller parts in isolation (unit tests) causes more testing to be done on the production cluster, where bugs are discovered later on in the development than what should have been necessary. With insufficient tooling, this quickly becomes a time-consuming and frustrating exercise; it is difficult to look inside the system and understand what is going on.

5.2.3. Build configuration

As can be seen in the table F.1, most packages in HLT did not make use of any build configuration tools in the beginning. Also important dependencies, like ROOT and AliRoot were in their early days lacking in this respect, but have since long had such facilities added. The effort to produce CMake files for HLT started in 2008 and was

5. Experience and performance

completed in 2011. This was before the shared source code repository and it was therefore difficult to have changes merged back into the original repository.

Build configuration allows for a portable code base that can be installed more easily to a variety of target environments. Not only for developers on their local machine, but also for deployment by administrators on clusters. If build configuration tools had been part of the HLT software from the start, it would have been easier to setup the development cluster in a nice and user-friendly way, so that it would have been more attractive to the users.

5.2.4. Shell scripts

Without a standardized build system, the process of compiling and installing all the software for HLT was automated by shell scripts instead - sometimes even in layers - making the process complex, opaque and unfamiliar to the developer. Shell scripts are also used in operator's wrapper-scripts (4.4.3) and as part of the input to chain configuration software (4.4.2.3). All these cases amounts to a considerable amount of shell scripts in the HLT repositories (see A.3).

Shell scripts are invaluable tools for the system administrator for automating tasks that are typically performed from the command line, but they are not a replacement for a proper programming language. While there are legitimate uses of shell scripts in HLT, they have also been the type of tool to turn to when having to patch together missing pieces during commissioning and there is no suitable infrastructure available that can simplify the implementation, nor is there enough time to properly design and implement a solution.

There is no clear candidate technology for a common, flexible infrastructure meant for communication between cluster software - something like a cluster bus or cluster api - that can easily be deployed in such a scenario.

5.3. Scope and methodology

The main goals of the evaluation in this chapter are to get a good idea about the general performance of the HLT project (project execution and end-product) and a better understanding of the more specific issues in the intersection between physics instrumentation, scientific computing and software engineering. An important aspect in this regard is the estimation of the effort involved in creating the HLT.

A recurrent theme in the literature on the interplay between scientific computing and software engineering is that there seems to be an unfortunate distance between the two. While there are programmer productivity issues in the various computational intensive sciences, little is lent from software engineering regarding methods, knowledge and experience that can help improve the situation [10]. Likewise, the fields studying scientific computing and software engineering in itself, have little contact with the application domain to ensure that the research that is being done really is relevant where it is meant to be used. This has created a chasm between the two camps [9].

5. Experience and performance

HEP instrumentation in the form of a high level trigger is a type of HPC and therefore belongs firmly to scientific computing. While the situation between scientific computing and software engineering is of great interest, there will also be value in looking at the HLT from the viewpoint of all involved fields and compare to research that treats those fields individually. To this end, good sources of data have been sought in project artifacts and relevant metrics have been identified.

Besides relevance, metrics have been selected while keeping in mind what can be accomplished in a satisfactory way as a single person contribution. This has in turn favored metrics that can be derived by static analysis of project/software artifacts. With static analysis, information can be extracted directly from the artifacts themselves, and third-parties need not be involved. Alternative, likely more involved approaches, could for instance be interviewing project members, performing code reviews of the source code and so on.

According to [75], a metric is a parameter that is observable and measurable. It is this definition that will be used here. The aspects that should be covered by a suitable evaluation are:

Physics For the physics part one can refer to published articles for documenting the performance or results achieved in HLT. Establishing further metrics for the physics performance is outside of the scope of this text.

Operational performance Certain metrics are relatively simple to collect, for instance those pertaining to the reliability of the operation, can be found by data mining of logs and monitoring systems. Also the performance of software components are interesting.

Project productivity For effort estimation, traditional software metrics like SLOC/CO-COMO will be used, but they will also be complemented with a HPC productivity analysis in order to get a more complete picture of the productivity in HLT.

Software quality Other metrics, like code coverage and code smells, that are relevant for code quality will also be included.

Together they make up a good set of metrics for measuring overall the performance of HLT and quantifying the experience of building the ALICE HLT.

5.4. Physics performance

Metrics for physics performance will not be extensively discussed, as these are thoroughly documented in published articles. But a short summary and references for the most important ones follows.

ALICE being specifically built for Pb-Pb collisions, the milestones of HLT are naturally tied to the timetable of the heavy ion runs. These are usually scheduled for a period of one month at the of the run period. The radioactive environment in the ALICE pit, and the extensive protection around the experiment, means that there is rarely access

5. Experience and performance

to do anything with the equipment inside the magnets during a run period. For ALICE this means that preparations for heavy ion runs needs to be completed typically almost a year ahead of time. For HLT, the situation is different since it is situated in a counting room further away from the experiment itself. Upgrades can therefore be performed in a more timely manner.

In September 2008, the LHC startup was underway and first collisions were expected to follow later the same year, but after only a few days of running, an incident [76] put the machine out of operation until late 2009. But, when first collisions finally happened on the 23rd of November 2009, HLT was contributing from the very start by calculating the vertex position of the collected events [77]. Later on, when beam conditions had stabilized, full reconstruction was performed on central barrel detectors and the muon arm.

In November/December 2010, a major milestone was reached as ALICE saw the first heavy-ion collisions [78], the type of collisions it was designed for.

While first heavy-ion in 2010 was exiting for ALICE, the heavy-ion runs in 2011 were particularly exiting for HLT as it was the first time HLT contributed to data that is stored for later analysis by performing selection and compression of reconstructed events [44][40][79][80].

5.5. Operational performance

After many years of prototyping and commissioning, first collisions in ALICE were finally observed in November 2009 [77]. Since then, HLT has been in stable operation, with increasing operational performance [40]. The preparations of the HLT team met its ultimate test with the heavy-ion runs in 2011 where events processed by HLT were included in the heavy-ion data that was stored for later analysis [44].

For evaluating the operational performance of HLT, there are several sources of information. DAQ provides a logbook [81] of all the runs with the most essential information from all the detectors and sub-systems. The DAQ infologger (4.4.4.2) that was adopted to be used in the HLT cluster records log messages from all the involved processes. Furthermore, the system logs of the operating system, the SysMES application and ESMP system, are informative sources, as are also the hardware devices (i.e. network switches) with SNMP capabilities.

5.5.1. Mean time between failures

Uptime or Mean time between failures [37] (MTBF) is often used about the life-expectancy of hardware components. As computer systems continue to grow and therefore also the number of contained components in the systems, the risk of a part failing rapidly increases [37].

HLT can be viewed from two sides in this context, both as a unit composed of smaller hardware components and as part of a greater whole, the ALICE experiment. The failures of hardware components in HLT have not been recorded in a sufficiently systematic

5. Experience and performance

way to allow for extraction of interesting statistics, but the entries in the logbook contains information about the length of a run and which detectors and sub-systems that participated, as well as the end-of-run reason. It should therefore be possible to calculate the MTBF of HLT, which would be a highly relevant metric for operational performance.

The weekly participation of HLT can be seen in figures 5.1 and 5.2. They show how long of the total run time that the HLT was participating.

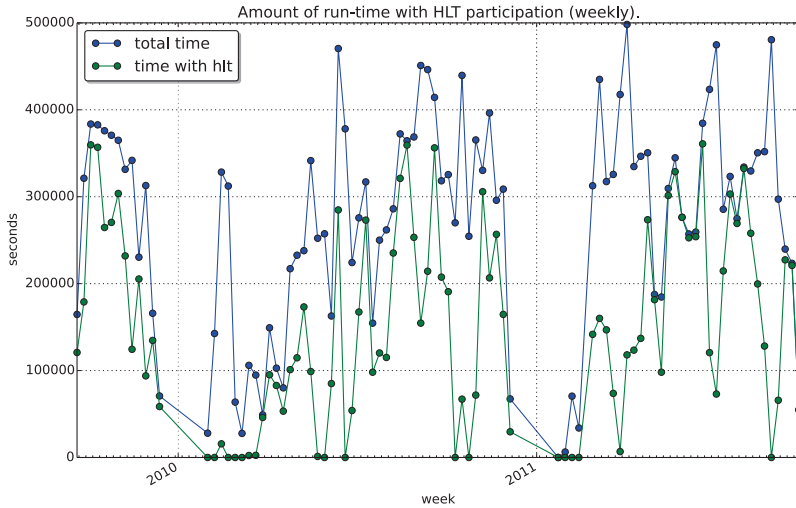


Figure 5.1.: Amount of run-time - in accumulated seconds - with HLT participation per week.

5. Experience and performance

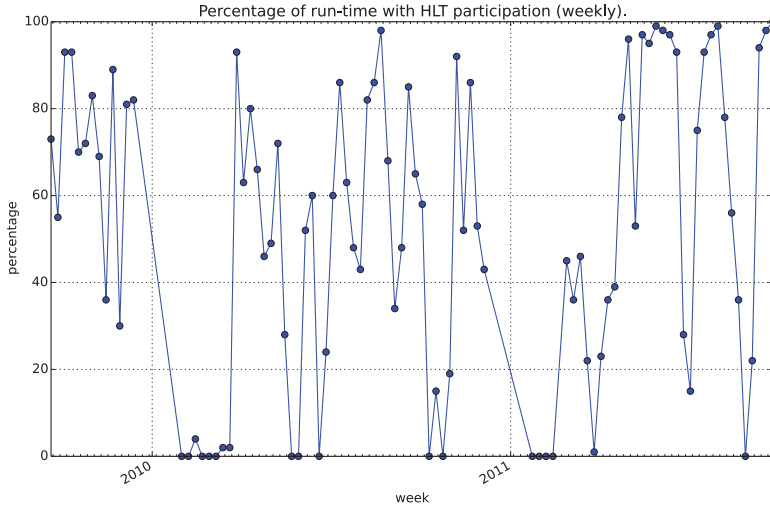


Figure 5.2.: Percentage of run-time with HLT participation per week.

There can be many reasons why HLT does not participate, and while taking HLT out of runs to perform testing or debugging still should be considered part of the “down time”, it is maybe more interesting to investigate how often HLT is the cause of a failed run or the reason why a run was stopped.

The “eor_reason” (eor: end of run) field in the logbook can be used as a quantitative measure of what causes runs to stop, although it is probably not entirely accurate as a portion of its input comes from shifters who might not always be able to determine the exact reason for a failed run.

Figure 5.3 shows the distribution of the “eor_field” field in physics (“partition = PHYSICS_1”) runs with HLT enabled (“HLTmode > A”). To the left all runs are shown, while only failed runs are shown to the right.

5. Experience and performance

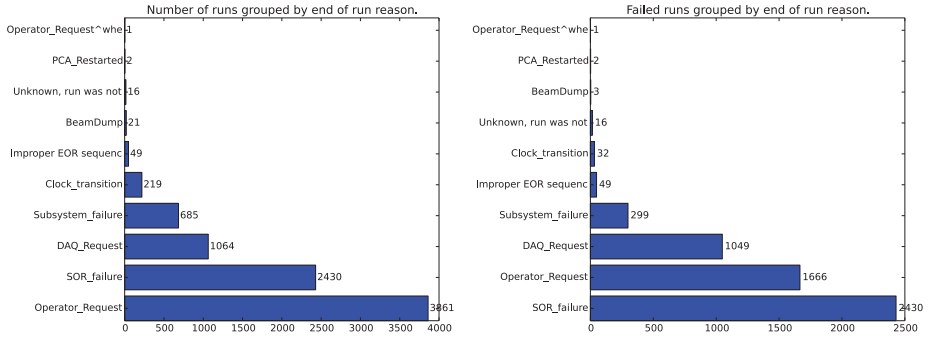


Figure 5.3.: End of run reason. All runs to the left and failed runs to the right.

The general conditions for a successful run is that it started and stopped without any issues in between. The two flags in the logbook that indicates success are: “ecs_success” and “daq_success”. Success is therefore here defined as when both “ecs_success” and “daq_success” are true. In any other case, the run is considered to have failed.

There are two bigger sub-groups of end-of-run reasons: “SOR_failure” and “subsystem_failure”. The distribution of these for failed physics runs with HLT can be seen in 5.4.

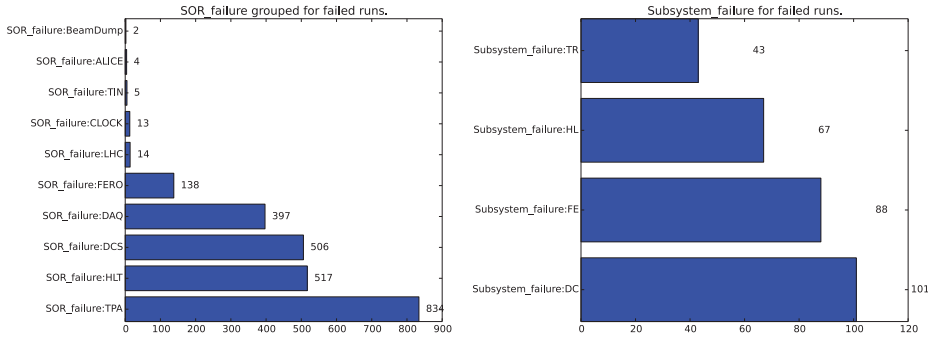


Figure 5.4.: Start of run failure to the left and subsystem failures to the right. The labels are extracted directly from the data. In the right figure TR means trigger, HL means HLT, FE means FERO and DC means DCS.

The SOR_failure value shown for HLT in figure 5.4, is the sum of all SOR_failure values that are associated with HLT. These can be broken down to their individual values as is shown in figure 5.5.

5. Experience and performance

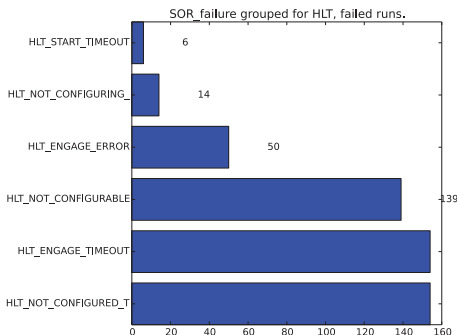


Figure 5.5.: SOR_failure for HLT only.

All in all, it is clear that HLT does not differ much in comparison to the other sub-systems in terms of operational performance. The numbers also show that apart from operator and DAQ requests, it is the startup phase that is the most critical. This is not surprising since this is when all the systems must be brought to a coordinated state.

5.5.2. Recorded log messages

The infologger system has been an invaluable debugging tool for the experts working with the HLT, but despite its usefulness, it also has some clear limitations. Being basically a centralized logging system, with a flat database, it offers not much more than existing solutions like syslog [82], except for some domain specific fields and a customized GUI.

The vast amount of log messages presented to the user in the InfoBrowser is large enough to be incomprehensible by the average shifter. Neither is there any assistance or intelligence built into the GUI that can aid the shifter in problem resolution, except for filtering by SQL-like syntax. The current setup should therefore be considered as purely an expert tool, which requires the presence of experienced personnel to be able to debug problems.

In systems with so many components in operation as in HLT, a demanding log stream is to be expected. But early on in the commissioning, superfluous messages from analysis components worsened the situation to the point where the infologger system could not keep up with the flood of messages resulting from a crashing chain, inducing long recovery times just for the infologger to soak up queued messages.

Since analysis components are initially mostly tested in isolation, one process at the time, without any rate requirements, it does not matter much if a few debug messages are emitted to better be able to verify the correct operation of a component. But when deployed on the cluster, replicated in hundreds of instances, running at rates up to 1kHz, the amount of a few debug messages per component will quickly add up.

Furthermore, lack of structure in the message format makes it hard to do any useful aggregation of messages and also makes pattern matching difficult, which in turn lessens

5. Experience and performance

the usefulness of infologger as a reliable input to recovery mechanisms. Although doable, complex data analysis is also hard to do on unstructured data. The graphs in appendix E shows what is possible to extract from the data today. To facilitate more advanced data processing, one should introduce structured logging for instance in the form of the proposed industry standard Common Event Expression [37] (CEE) [83].

In conclusion, experts are required to be available on-site during operation in case debugging is needed with the currently available tools.

5.6. Effort estimation and development performance

Despite programs being created to be executed by computers, the process of developing software in larger projects is very much a human activity. The human aspect and the intangible nature of software makes estimation of scope and effort inherently hard (see for instance [84]).

For a systematic approach, one can make use of established techniques for software cost estimation. These can be based on expert judgement or they provide algorithmic models that will produce estimates based on some input. An overview and comparison of several estimation techniques can be found in [85].

Algorithmic models mainly rely on software metrics and tuning of input parameters for producing estimates. This can be frustrating during early development, when many of the most useful metrics are not yet readily available (i.e. measuring lines of code is impossible up-front), but for projects nearing the end of its development, metrics can be extracted from project-artifacts. Techniques based on arithmetic models are therefore a good fit for retrospective analysis, such as is the case for estimating the expected effort involved in building the HLT. For making predictions about program size and involved efforts beforehand, a method like Function Points would be better suited, as it is based on information derived from the requirements [86].

While approaches that looks at OO attributes of the source code have seen increased focus following OO becoming the predominant programming paradigm, some studies question the validity of such approaches and suggests that they may not be any better than pure size metrics [87]. So while Source Lines of Code (SLOC) - which is discussed below - might seem like a primitive metric, it might still be as useful and relevant as any other current metric, as others fail to improve upon it.

When evaluating a complex system such as HLT, one should consider the entire enterprise. Everything from management and human processes, to software and release engineering, system administration and of course also the physics aspect which the HLT is built for in the first place.

A relatively recent field of study that touches upon many of these points is HPC productivity [88]. Together with more traditional, but related concepts, it would be very useful for producing a better productivity measure of HLT.

5.6.1. Source lines of code and the Constructive Cost Model

Constructive Cost Model (COCOMO '81) is an algorithmic cost estimation model for software introduced by Boehm in his book *Software Engineering Economics* in 1981 [84]. He derived his model by looking at data from several projects, identifying a set of attributes that seemed to affect their duration, staffing and cost.

These attributes or parameters are used to tune the model, which main input is SLOC, a common size metric used in software estimation. COCOMO therefore leans more towards static analysis of artifacts when compared to other approaches like Function Points or expert judgment.

COCOMO comes in three increasingly more complex forms; basic, intermediate and detailed. The basic model will be sufficient for the usage here as the goal is only to get a rough estimate. Estimating the cost drivers needed for the more complex variants is often of little gain and can even make the estimates less accurate [89]. With the basic model, the effort and the schedule parameters can be tuned according to the type of project as described in the SLOCCount User's Guide [90]:

- Organic: Relatively small software teams develop software in a highly familiar, in-house environment. It has a generally stable development environment, minimal need for innovative algorithms, and requirements can be relaxed to avoid extensive rework.
- Semidetached: This is an intermediate step between organic and embedded. This is generally characterized by reduced flexibility in the requirements.
- Embedded: The project must operate within tight (hard-to-meet) constraints, and requirements and interface specifications are often non-negotiable. The software will be embedded in a complex environment that the software must deal with as-is.

From these, embedded seems to be the best fitting description of HLT software.

The remaining consideration then, is if the hlt-alice repository should be considered one big monolithic project or as several sub-projects, one per top-level directory. Table 5.1 shows the four possible combinations of organic/embedded and multi/mono project. An overview with key data of the various repositories used by HLT is shown in section 4.4.1.

5. Experience and performance

	Person Developers	Person Months	Person Years	Schedule Months	Schedule Years	SLOC
mono/embedded	177	7811	651	44	4	603040
mono/organic	44	1993	166	45	4	603040
multi/organic	62*	1775	148	28	2	603040
multi/embedded	176*	5010	417	28	2	603040
subprojects	275	4954	413	242	18	593132

Table 5.1.: COCOMO variants of the full hlt-alice repository together with an entry where all subprojects are summed up. The numbers marked with an “*” was not produced directly by sloccount, but summed up over the individual project results. COCOMO numbers for external packages used in HLT can be found in the appendix A.2.

The multiproject setting just affects how the calculations are being done, it does not produce individual COCOMO results per sub-directory. Breaking the numbers further down by considering sub-directories individually, improves the per project accuracy and can facilitate more interesting analysis. Key numbers can be seen in table A.1.

Immediately these numbers seem higher than one would expect, and in fact, at least one study has shown that in many cases, the COCOMO model is biased towards higher-than-realistic numbers and goes on to explain this by arguing that COCOMO as presented by Boehm is tied too closely to the environment from which it was originally derived; aviation software with most likely stricter requirements than typical business applications [89]. This must be kept in mind when these numbers are later used for comparisons.

5.6.2. Developer activity

Even by a cursory glance, the number of commits over time, as visualized in figure 5.6, gives a good indication of the various phases in the HLT project. First, there is a long prototyping stage, starting around 2000 lasting until development ramps up in 2005/2006, reaching a peak in 2008. Following years have high, but declining activity until reaching a maintenance state in 2012.

5. Experience and performance

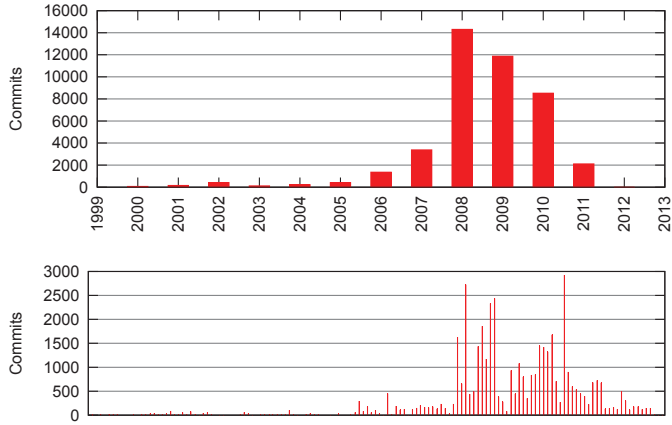


Figure 5.6.: Commit activity by year (top) and month (bottom) as produced by gitstats [91]. Shows number of commits over time for the whole hlt-alice repository.

This corresponds well with the physics milestones mentioned in section 5.4; first collisions were initially planned for 2008, but got delayed until 2009. First heavy-ion collisions happened in 2010 and in 2011 HLT contributed to the triggering in ALICE as well as compression of reconstructed events. From 2012 and onwards, the project enters a stable phase where most functionality has been delivered and the activity winds down accordingly.

Keeping in mind that repository statistics is purely a programming artifact, one can still relate the phases and activity to those found in Rational Unified Process (RUP) [92] and depicted in figure 5.7. Although one should consider early coding and prototyping to be part of the analysis and design activity, and similarly the later maintenance work to be part of the deployment, while the mapping of implementation to the commit activity is more direct.

5. Experience and performance

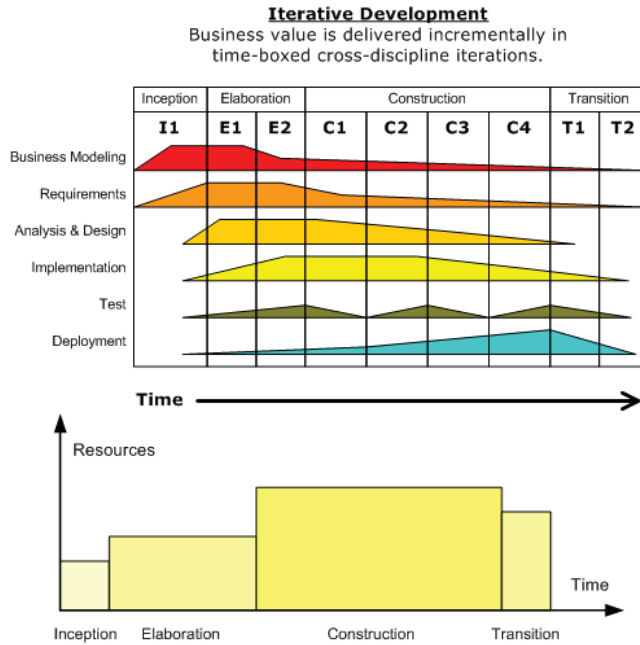


Figure 5.7.: RUP phases and activities at the top [93], and a typical project profile below [94].

Further detail and per-directory statistics can be produced by parsing the commit logs directly from the repository. It then becomes easier to extract activity profile for each module that can be compared to SLOC/COCOMO to see if there is any correlation. Key numbers for commit logs are listed in appendix B.1 and table 5.2 collocate numbers from both COCOMO and commit activity.

5. Experience and performance

	months	commits	devs	changes	SLOC	devs	months
hlt-alice	150	11611	27	4627306			
BCL	130	175	2	53526	20731	11	137
Components	88	795	2	214878	82884	35	722
EVTREPLAY	0	6	1	1191	721	1	2
Framework	122	426	2	501069	81127	35	704
H-RORC	16	57	1	1370405	192976	70	1990
MLUC	128	237	4	66773	26898	14	187
PSI	69	109	2	47933	6313	4	33
PSI2	64	35	3	16904	8112	5	44
RFLASH	16	8	1	2173	1252	1	5
RunManager	16	32	1	7967	3320	3	15
SimpleChainConfig1	93	285	2	40818	13784	8	84
SimpleChainConfig2	47	207	2	25773	11777	7	69
TPC-OnlineDisplay	0	3	1	4357	571	1	2
TRD-Readout	1	2	1	2638	2247	2	10
TaskManager	104	155	2	25921	13223	8	80
Utility_Software	82	228	3	35170	20231	11	133
control	56	6433	14	686602	38259	19	285
interfaces	41	62	4	208019	38740	19	290
monitoring	9	19	2	4697	2239	2	9
restricted	39	89	5	14617			
tools	54	245	4	40256	9111	6	51
trunk	81	281	1	40783	13765	8	84
util	57	8	1	2700	1193	1	4
utilities	81	61	4	20696	2441	2	10

Table 5.2.: Comparing COCOMO numbers (right of vertical line) to commit activity (left of vertical line). Months for commit activity is the number of months between the first and last registered commit. Changes is the number of lines that has been changed in the commits.

It is difficult to draw any definite conclusions from these numbers. Author and developers corresponds here to the same thing, but timespan is just a rough estimate of the time spent on a project, as work - as far as it is reflected by commits - might appear in varying intensity over time. This is illustrated in the commit activity diagrams, which shows how the activity of the various sub-projects spreads out in time, how they overlap, and how authors worked on multiple related projects simultaneously. They give a good at-a-glance impression of the development.

5. Experience and performance

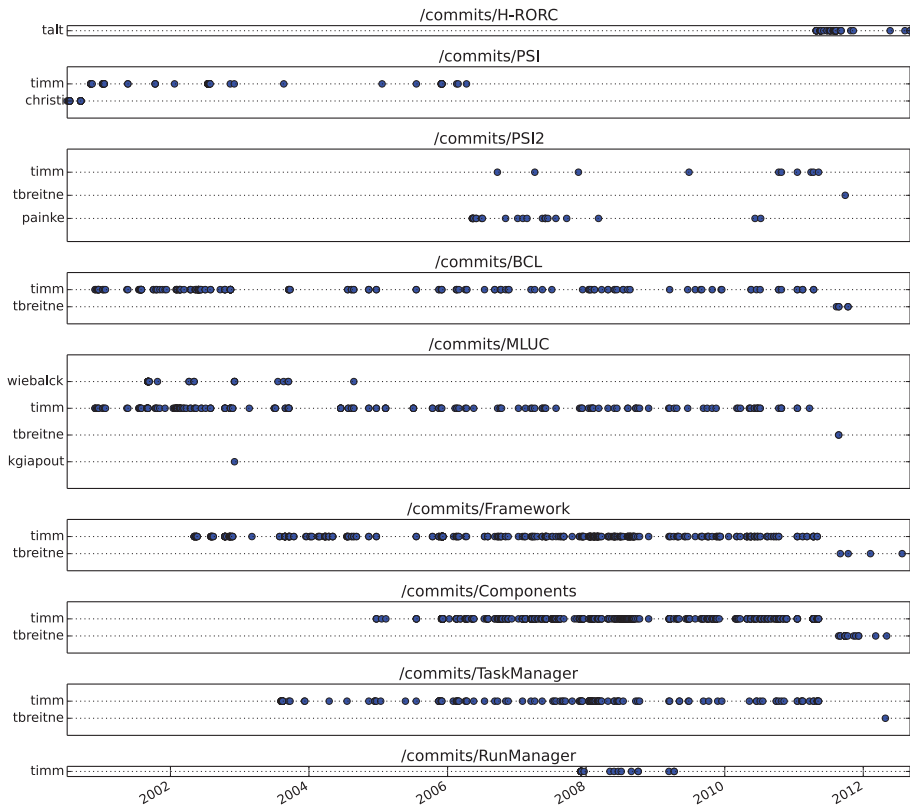


Figure 5.8.: Diagram of commit activity per project over time, showing run-critical projects. One dot represents a commit in time for the developer listed to the left.

5. Experience and performance

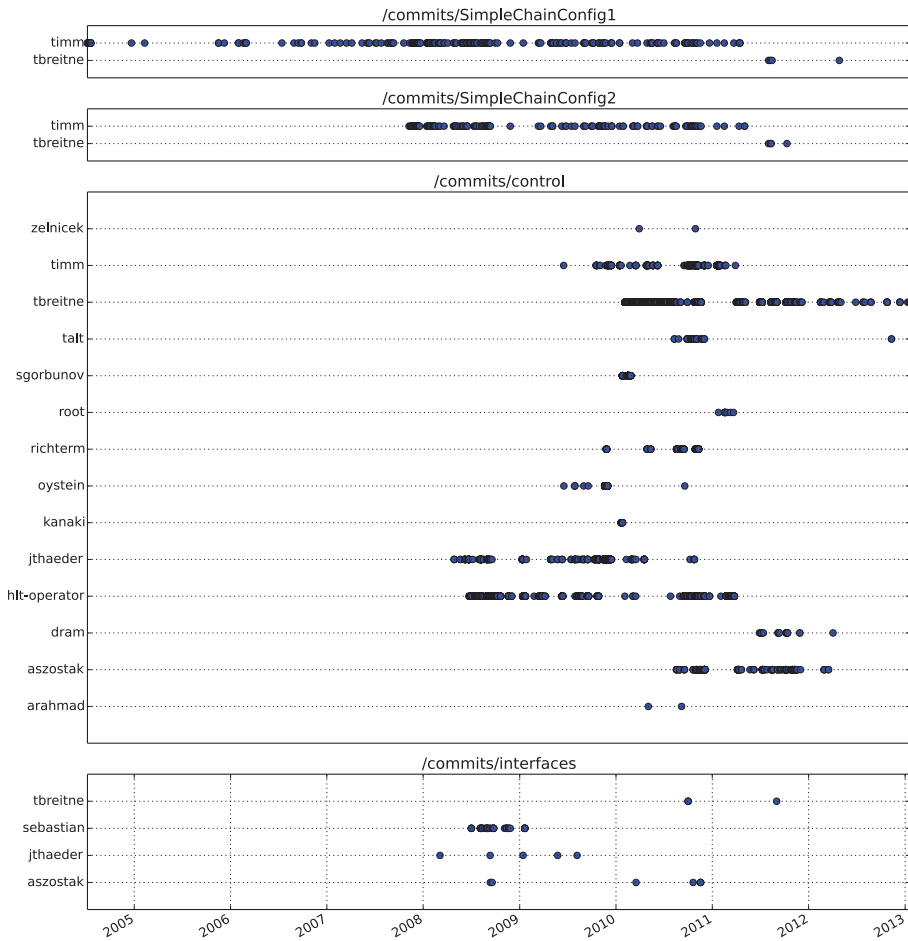


Figure 5.9.: Diagram of commit activity per project over time, showing projects related to operation. One dot represents a commit in time for the developer listed to the left.

5. Experience and performance

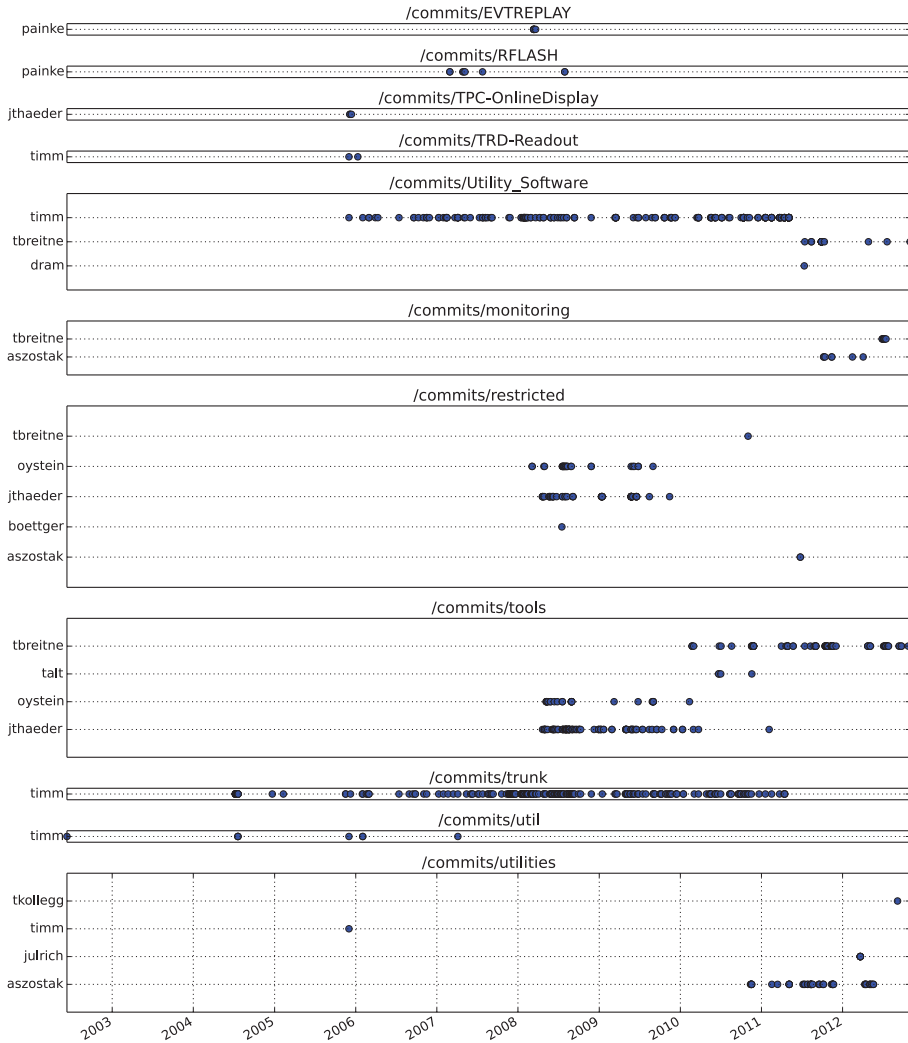


Figure 5.10.: Diagram of commit activity per project over time, showing the remaining projects in the main HLT repository. One dot represents a commit in time for the developer listed to the left.

5.6.3. HPC productivity

Computational performance, as measured by rate of performed operations, has for a long time been the only benchmark applied to HPC, as exemplified by the top500 list [95] and its accompanying LINPACK test suite [96]. But as data center usage continue to rise and starts to consume non-negligible amounts of energy (about 1.5 % of the world total in 2006 [97]), power efficiency in HPC is seeing increased attention, to the point that there is now a green500 list where the ranking criteria is performance per watt [98].

Lately, an even broader meaning of performance in HPC has emerged as HPC productivity. A model synthesis that includes several more criteria for benchmarking the overall productivity in a HPC cluster, has been derived in [75]. It incorporates cost, program size, execution efficiency and programming language expressiveness into a small framework for productivity analysis.

In this model there is a clear relation between program size and project cost. The seemingly unhelpful and obvious insight that the best way to decrease cost in a project is by reducing the number of source code lines produced, have some interesting implications; reuse of existing software and expressiveness of programming language should be very interesting topics when discussing how to decrease cost.

The definitions of several important concepts in HPC productivity is given in the introduction of [75]. Here, productivity is defined as utility over total cost and expressiveness is said to quantify the abstractness of a programming language.

5.7. Software quality

It is the packages that are critical for operation, as indicated in table 4.2, that will be included in the following discussion that attempts to get an impression of the state of the software through the use of software metrics.

5.7.1. Technical debt

As a software system grows, it is only natural that the idea of its ideal structure changes as the knowledge about its function increases. When a feature is about to be added, the developer is often faced with a dilemma: should the existing code base first be modified to fit a new and better understanding or should the feature be added in a possibly sub-optimal way?

It is when features are added without regard to maintaining internal structure that source code can deteriorate. Over time it will become increasingly harder to make modifications to the code. Technical debt [99] is important as an analogy for developers to communicate why one should continuously pay attention to improving the structure of source code so that the software stays maintainable and so that it is easy to add features in the future.

Technical debt is not only about the sources code itself, but also supporting artifacts. In HLT the lack of documentation and tests for large parts of the software is a clear sign of technical debt.

5.7.2. Code coverage

Testing is crucial for producing robust code. A typical metric for code quality is code coverage, which is the ratio of executed lines over executable lines that have been exercised during tests in a project [100]. It is usually produced while running unit tests. Unfortunately there are very few unit tests in HLT software. The transport framework, the most crucial part of HLT, has none. Without unit tests and a test framework one cannot easily work out the test coverage.

There are however some test programs and scripts spread throughout the source repositories. From these one can still try to find the ratio of lines of test code to lines of source code, by sorting files in two sets; test files and code files, with the sorting criteria being whether or not the word “test” is part of the file’s path or name. This is an ad-hoc approach that relies on common naming convention for getting the sorting right.

The results presented in tables 5.3 are quite far off from the suggested rule of thumb of about 1:1 for Lines of Code (LOC) to Lines of Test (LOT). The relevance of this metric is often debated though, as can be seen in discussions among members of the software engineering community [101].

	lot_ratio	code_duplication
alroot:HLT	0.00	0.26
hlt-alice:BCL	0.15	0.41
hlt-alice:Components	0.86	0.60
hlt-alice:Framework	0.04	0.34
hlt-alice:H-RORC	0.00	0.32
hlt-alice:MLUC	0.55	0.48
hlt-alice:PSI2	0.00	0.41
hlt-alice:RunManager	0.05	0.70
hlt-alice:SimpleChainConfig1	0.00	0.45
hlt-alice:SimpleChainConfig2	0.01	0.20
hlt-alice:TaskManager	0.04	0.25
hlt-alice:bigphys	0.00	0.00
hlt-alice:control	0.01	0.62
hlt-alice:interfaces/DDCTestGui	0.00	0.37
hlt-alice:interfaces/HCDBManager	0.00	0.13
hlt-alice:interfaces/ecs-proxy	0.06	0.08
hlt-alice:interfaces/fed-portal	0.09	0.19
hlt-alice:interfaces/pendolino	0.04	0.24
hlt-alice:interfaces/taxi	0.00	0.04
hlt-alice:trunk	0.00	0.45

Table 5.3.: Software quality metrics for HLT software.

Although not very reliable, these numbers can be seen as indicators that the conditions and tools that foster high quality software have not been in place during HLT develop-

5. Experience and performance

ment. They at least tell us that the testing situation is abysmal; it is not even possible to get a reliable measure of how large portion of the code has been tested.

With low test coverage, more time is spent on manual testing and debugging since problems are to a lesser extent caught during development, and more bugs also makes it into the production system. Lack of unit tests in HLT software can also have been a contributing factor to the low collective ownership, as it is harder (time-consuming to check others code) for project owners to trust modifications from other developers to be bug free without a proper test harness in place.

The amount of testing in a project is only an indirect indicator of software quality. More direct indicators would be how time-consuming debugging is, how hard it is to add new features, how fragile the system is during operation, etc. Some of these qualities are also known as design smells [102]. They are symptoms of design principles having been violated.

5.7.3. Code smells: code duplication

Effort estimation as done in section 5.6 is only scratching the surface of the state of software in a project. Under the hood, it is interesting to see if it is possible to get a better impression about the quality of the source code by looking for code- or design smells. Code smells are signs of bad software design that can make it difficult to work with the source code as is. Discovery of code smells should normally trigger a refactoring before any new code is added.

Most smells are about keeping the logical elements small and coherent; methods and classes should not be too big and they should mainly be doing one thing. Code duplication is one of the smells as defined by Beck and Fowler in their book on refactoring [103]. It is easy to discover with static source code analysis and removing it carries a high reward.

Several tools exists that can be used to extract this number from source code, see for instance [104] and [105]. Simian has been used to produce the numbers for HLT that can be seen in table 5.3.

It shows that the code duplication in the core software of HLT is quite high. In a survey of research in clone detection, the percentage of code duplication for typical software systems lies within the range of a few percent and up to around 20% [106]. Most HLT packages have numbers above this higher bound.

5.8. Summary

The very nature of large experiments makes them unpredictable and difficult to plan as they are always about learning while building something new. And when doing something for the first time, one cannot easily rely on past experience. There are however plenty of opportunities to borrow from similar experiences and practices from other fields.

Many lessons can be learned from the installation, the commissioning and the upgrades of the HLT cluster. As always, the right amount of qualified and experienced people, at the right time and place, is crucial for smooth operation and a successful project. These are challenges shared with any other collaboration in experimental physics. Nevertheless,

5. Experience and performance

framing these general observations in the context of HLT, can provide useful insights for similar future projects.

The HLT project has succeeded in fulfilling its functional requirements and has delivered a High Level Trigger satisfying the expectations of the ALICE experiment. Still there are elements of the original design documents that are not reflected in the current implementation. Deviations are to be expected due to the evolving nature of an experiment, something also acknowledged by early design documents, but also problematic design and project management can lead to friction in project execution, resulting in unforeseen changes, reduced scope and a struggle to deliver what was promised.

The real-time redundancy/failover in the processing chain is one example of something that has not been completed, although it was initially considered an important feature for reliable operation of the HLT and the transport framework had been prepared for it.

Similarly for the management solutions, one can say that they have not yet become as sophisticated as they initially were intended to be, in that they still require quite a bit of configuration and adoption to the site where it is installed to perform their tasks.

From the software development point of view, one can see some signs of code smells in HLT, as has been mentioned earlier in this chapter 5.7.3. In particular code duplication seems very high. A likely cause of such smells is the general lack of supporting artifacts (unit tests, documentation), and to some extent, infrastructure in HLT:

- The low amount of structured, automated tests in HLT makes it difficult to leverage modern software engineering practices. This is unfortunate as the complexity of HLT could really benefit from such practices to confidently be able to guarantee a robust end-product.
- Proper build configuration is not only important for portability, but also in that it allows for further automation. Most importantly continuous integration and packaging. The introduction of build configuration was much too late in the project to be as useful as it could have been.
- Hardware-wise, the infrastructure for the development and commissioning has been good. But, it has been difficult for users to make full use of the development cluster due to immature software and lack of configuration, as the main cluster required most of the attention from the administrators.
- A common shared source repository was added when implementation was mostly done and commissioning well under way, too late in the project to be able to get the full benefit from using it.

The lack of good tools for debugging large, distributed systems, a common challenge in the HPC industry, also affects a project like HLT. The real-time nature of HLT adds considerably to this challenge. Getting involved in ongoing research efforts that seeks to improve the tooling situation would be one way of becoming more proficient in solving problems in the HLT cluster. The few tools that are available today are mentioned in the next chapter 6.6.3.

5. *Experience and performance*

Improved tooling extends to logging and statistics, where structured logging would be a welcome improvement, as would better ways to unify and correlate information from all sources in a cluster.

For most of these issues it is tempting, and probably correct, to conclude that the scope of the project has not been met with sufficient resources. But, it is maybe precisely for this reason, that one would expect a greater eagerness to make use of practices that can aid the project towards a steady, predictable process, increasing the quality and value of the end-result.

Up-front design and lack of a process for keeping design and implementation in synchronization with a changing reality can lead to wasting a lot of precious resources for implementing elements that might not even be needed in the end.

The relatively large portion of scripts (see A.3 and 5.2.4) can be seen as an indication that the effort of commissioning the HLT software in the production system was underestimated and that the overall plans were lacking in aspects related to end-user interaction and software management.

The upshot is that although the system is working fine today, the HLT was brought to its current state through a lot of debugging and laborious integration work, that could have been reduced by making greater use of established practices. It is rarely a question of if - normally the needed software for an experiment will be made to work sufficiently well - the question is for what price? What could have been saved if SE was successfully applied in HLT?

Furthermore, lack of artifacts, like documentation and automated testing, is going to make it difficult to build on, and extend the current code base for future projects. This is an unfortunate outcome considering the potential savings.

The experience in HLT resonates well with research that suggests that knowledge form software engineering is not being picked up by the scientific computing community [10], as discussed in chapter 2.

6. Possible improvements

The time and money invested in the experiment makes for a strong incentive to keep the sub-systems of the experiments in operation for as long as the machine is running and there is an opportunity to collect data. Increasing reliability and maintainability in compute clusters is crucial in delivering high quality services throughout the lifetime of a typical HEP experiment.

At the same time, the complexity of real-time high-performance systems like the HLT in ALICE makes reliable operation a challenge that requires personnel on-site to keep the system running. A more self-managed system would reduce the reliance on people on-site and thereby also operational cost. Beyond the functional requirements, these are some of the most important motivational drivers when designing and deploying a HPC cluster in HEP.

While a “holistic” approach is useful for having the best understanding of a system, it is also important to set the appropriate scope for the work discussed in the remainder of this text. The focus will from here on mainly be on the software aspect of HLT, in particular for the non-core applications, and it will be from a cluster administration and software engineering point of view.

This chapter will start by reviewing current trends, techniques and ideas that can be useful in improving the HLT. The challenges experienced in HLT are the same as the computer industry at large is facing these days; the increasing difficulty of handling the complexity in large computer systems. As a result, topics that deals with this aspect will be of particular interest.

6.1. The autonomic computing initiative

One way of handling complex systems, is to make each and every constituent part smarter, more aware of itself, more self-contained and more autonomous [107]. The autonomic computing initiative from IBM is maybe the most well-known attempt at achieving this. In the manifest from 2001, their vision was announced:

IBM has named its vision for the future of computing "autonomic computing." This new paradigm shifts the fundamental definition of the technology age from one of computing, to one defined by data. Access to data from multiple, distributed sources, in addition to traditional centralized storage devices will allow users to transparently access information when and where they need it. At the same time, this new view of computing will necessitate changing the industry's focus on processing speed and storage to one of developing distributed networks that are largely self-managing, self-diagnostic,

6. Possible improvements

and transparent to the user [108].

According to IBM, a software system adhering to this paradigm, must exhibit these fundamental properties from a user perspective [108]:

- Flexible. The system will be able to sift data via a platform- and device-agnostic approach.
- Accessible. The nature of the autonomic system is that it is always on.
- Transparent. The system will perform its tasks and adapt to a user's needs without dragging the user into the intricacies of its workings.

The autonomic computing manifesto [108] then goes on to define 8 elements (later reduced to four: Self-configuring, Self-healing, Self-protecting and Self-optimizing) that characterize autonomic computing. This text is concerned mainly with the points 1-3 and to some extent 7 (about favoring open and non-proprietary solutions), as these are the most relevant for HLT:

1. An autonomic computing system needs to "know itself" - its components must also possess a system identity. Since a "system" can exist at many levels, an autonomic system will need detailed knowledge of its components, current status, ultimate capacity, and all connections to other systems to govern itself. It will need to know the extent of its "owned" resources, those it can borrow or lend, and those that can be shared or should be isolated.
2. An autonomic computing system must configure and reconfigure itself under varying (and in the future, even unpredictable) conditions. System configuration or "setup" must occur automatically, as well as dynamic adjustments to that configuration to best handle changing environments.
3. An autonomic computing system never settles for the status quo - it always looks for ways to optimize its workings. It will monitor its constituent parts and fine-tune work flow to achieve predetermined system goals.

To introduce autonomic computing in HLT, a natural place to start would be to increase the self-awareness of the cluster and its software. For that there needs to be in place a mechanism for gathering information and making it available to interested parties.

Information about the cluster could either be gathered up front to a central place, queried on-demand, pushed on change or by a combination of any of these. For distributed applications it is convenient if information is available on the very same machine as the participating process is running. But for processes that operates more globally on the cluster, such as data mining, extraction of statistics, and similar, it might be desirable to have dedicated instance for the heavier queries and hence a centralized service to complement a distributed one.

This is very much in line with the point of transparent access to data mentioned in the vision above, and also aligns well with the requirements of HLT; the need for resource management, which in turn needs an inventory database as one of its inputs.

6.2. Current trends and relevant concepts

The technological advancement has a tendency of finding its own way, independent of grand visions, and it is therefore important to also understand the current trends. Many topics in computer science and software engineering embodies techniques and approaches that HLT could benefit from. What will be mentioned in this section is in no way exhaustive, but includes the fields considered to be most relevant.

6.2.1. Increased performance of dynamic (interpreted) languages

The concept of high level languages has been with us for a long time, although what has been considered a high-level language has changed over the years. Languages that we today may think of as low-level - like for instance C - were earlier considered high-level to their contemporary alternatives.

In software engineering, the notion that the productivity in terms of lines of codes per time unit is constant, independent of programming language, has been a long hold rule of thumb, that has also been verified in smaller studies [109]. This implies that a language with high expressiveness can implement more functionality than a language with lower expressiveness in the same time period, something that is reflected in function point to SLOC translation tables [110]. It also means, as mentioned in section 5.6.3, that when project cost is tightly coupled to the size of a program, expressiveness becomes a very interesting criterion when evaluating programming language for use in a project. The difficulties of weighting expressiveness and performance in HPC are discussed in [75], where also a formula is given that can help in deciding when a high-level dynamic language is or is not the most appropriate choice.

Due to their productivity and ease of use, dynamic, high-level languages have seen increased popularity over the years [111]. The high level of abstraction makes it easier to address larger conceptual parts at the time and thereby allows faster development of complex software. With less actual code, there is also less room for making mistakes, an observation that has spurred the suggestion that there is a rather strong connection between SLOC and number of defects [17].

Programmer productivity is further helped if the language is interpreted, which is often the case with dynamic languages. Without the compilation step, interpreted languages have shorter development cycles, and a more exploratory style of programming is possible as the threshold to try new approaches is low, and the feedback is in most cases instantaneous.

A dynamic programming language allows programs to change their code and structures during runtime, enabling certain ways of expressing functionality not possible with static languages [111]. Usually a dynamic language will also be dynamically typed, and conversely, a static language statically typed. But the difference between the two lies mainly in how type checks are performed. Statically typed languages performs type checks during compilation, while dynamically typed languages does type checking during runtime.

For this reason, static typing can help catch a certain class of bugs early on during

6. Possible improvements

compilation that otherwise would only have surfaced during runtime with dynamic typing. With proper test coverage though, which is important independent of language type, there would in practice be little difference, as the most important code paths would have been exercised during testing, catching the bugs before being put into production. Also, by using a modern Integrated Development Environment (IDE), syntax highlighting and code completion will help catch syntactical errors and ensure correctness while code is being written, further reducing the risk of introducing defects.

The advantages of higher-level code are therefore well-known as are the disadvantages; it has always been about weighing development cost against execution efficiency. While the productivity of dynamic/high-level languages is rarely disputed, the association with interpreting makes performance the most often heard concern about their use when compared to compiled languages. Normally interpreted programming languages are fast enough for most applications, but the performance sought in HPC have traditionally warranted the use of Fortran, C or C++.

But what we might be witnessing in recent years, is that the current generation of high-level languages are being treated to classic (language transformation to compiled language) as well as more recent approaches (just-in-time compilation) to enhancing performance, while at the same time a flourishing developer community has sprung up, building highly efficient data structures and data handling libraries (PyTables, Numpy, Pandas are just a few examples for python) for these languages.

While high-level languages have for a long time been part of the scientists and the HPC builders toolbox, these recent changes effectively reduces the traditional disadvantages to the point where high-level languages might soon be a viable alternative also for computational-intensive tasks.

The next logical step in the strive towards even higher-level languages, for reasons of even higher programmer productivity, is maybe already seen in declarative extensions being added to popular libraries and programming frameworks/languages.

With declarative languages, the programmer states what the program should do and the runtime environment or compiler takes care of the details of how it should be done [112]. One of the goals of declarative approach is to avoid side-effects; the application developer is confined in an environment where he can easily articulate his problem without having to worry about implementation details and getting those details right.

These are on the other hand taken care of by the language designer or library developer, offering a nice separation of work where both professionals can invest their effort and use their talent where it makes the most sense. These benefits extends to the source code itself, where separation of concerns, is since long, generally considered to be a good practice in software construction [113].

The QML language of Qt [114][115] is a good example. It lets the user define UI elements, transitions and states in a simple JSON-inspired language, while behind the scene, there is a hardware accelerated scene graph that translates the high level declarations into efficient execution [116].

Object storage layers, like ORMs, are another area where declarative extensions can be found. Normal object definitions are here augmented by decorators or hints that enables the declarative layer to mirror live objects to persistent storage and keep them in sync,

6. Possible improvements

without the developer having to write any storage layer code in between. The declarative extension of SQLAlchemy [117] serves as a good example of how this can be done.

As for scientific computing and massively parallel applications, research is ongoing to see if one can apply declarative programming to these contexts as well, and to see if it will lead to programming systems that makes it easier for the scientist to harness the power of multicore machines [118].

6.2.2. Growth of Free and Open Source Software (FOSS)

Sharing of source code has been a natural part of programming communities since the inception of software. The ideals of sharing, transparency and peer-review in Free and Open-Source Software (FOSS) are the same as those science are built on. This might be the reason why FOSS is so wide-spread in scientific communities.

It still took a while before the FOSS concepts we know today found its final shape and was formalized through creation of licenses [119], such as those approved by the Open Source Initiative [120] and the Free Software Foundation [121].

Later, acceptance in the commercial and business world have gradually followed until today when many of the largest enterprises are building their fortunes in the cloud on top of FOSS and mobile devices are increasingly being powered by more or less open operating systems (android). To more and more businesses it makes sense to take part in the collaborative development of commodity software, such as operating systems, web stacks, browsers and office suites, and then compete with offerings that builds value on top of that [122]. By participating, one can influence the direction of commodity software, while the development cost is shared amongst the collaborating parties.

FOSS, being free as in gratis, makes it a great enabler of code reuse. Studies shows that code reuse is at least as frequent in FOSS as in proprietary software [123]. Quality is also generally on par with proprietary software, as shown for instance in the yearly reports by coverity [124], a firm specializing in static code analysis.

The strongest point of FOSS though, is maybe the flexibility that comes with availability of source code and thereby being able to make changes in the entire software stack, and being able to mix and match a plethora of software packages freely, to get as close as possible to the wanted solution.

6.2.3. Clusters, the Grid and Cloud

A cluster is a collection of two or more locally connected computers, configured and equipped to behave as one (bigger) coordinated machine with the purpose of improving one or more properties of its operation - be it availability, concurrency or raw performance. Clusters are commonly used in science to reduce the processing time of simulations and data analysis by running work-loads in parallel.

When geographically dispersed clusters are interconnected to form an ever bigger computing infrastructure, it is called a Grid, taking its initial namesake from another pervasive infrastructure; the power grid [125]. Grid was introduced on the advent of LHC, when it was realized that it would not be possible to host all the needed computing re-

6. Possible improvements

sources at CERN alone [25]. Both Grids and clusters often behaves as one big application or system, performing massive tasks for many users, usually for the same organization or field of study.

Cloud can be thought of as the unification of physical nodes and infrastructure - so in ways similar to clusters and the Grid - but equipped in such a way as to allow for its resources to be split up in virtual computing units of arbitrary size. Its ability to easily provision virtual resources makes it ideal for commercial usage where the same infrastructure can host services for all kinds of businesses in all kinds of sizes.

While cloud is for now mostly available as commercial offerings - either in the form of services or infrastructure - solutions like openstack [126], can be used to build private cloud infrastructure. Started by rackspace and NASA, openstack is free to be setup by cloud providers and private companies alike, giving the benefit of having a common cloud infrastructure in both domains, as well as being vendor independent.

This should be interesting also to the scientific community in the near future, perhaps not for core computational and long running operations, but rather for workloads and usage patterns that are more transient, and for handling peak loads when needed, offloading existing Grid infrastructure. A sign of such interest was the recent announcement that Rackspace would join openlab at CERN to collaborate on openstack [127].

6.2.4. Loose coupling and the resurgence of REST

It is generally considered good practice to have a high internal cohesion and low external coupling in software components [128]. In this context, cohesion means that the functionality contained in a component is to a large extent related, while coupling is the degree of reliance of a component on the internals of another component.

Components of loosely coupled systems can more easily be developed independently from each other, allowing a more flexible and rapid development. This is one out of several scalability parameters in distributed systems. The concept of an API can serve as an example of a method that helps decouple software components. The main purpose of an API is to hide implementation details for client code, making it easier to reuse software components; the underlying implementation can be changed/improved at any time, as long as the API itself remains the same.

Loosely coupled can also denote an architectural style that makes it easy for a system to scale out processes to compute nodes, a feature naturally favored by distributed systems such as clusters, Grids and cloud. Cloud itself and in particular SaaS - Software as a Service, are examples of how successful loosely coupled systems can be in helping to massively scale software in the market today.

For distributed computer systems to have any meaningful operation, they will internally have to agree upon a scheme for how information should be exchanged. In applications that make use of a client-server architecture, this is typically accomplished by defining a communication protocol that must then be implemented by the software that is to be compatible.

Software protocols are similar to the conventions of normal conversation. The initiator of the exchange, usually the client, announces its intentions to interact with the server

6. Possible improvements

and will subsequently get a reply. There might be a hand-shake and authentication of the user that acts through the client. Then requests or commands are sent from the client to the server and responses are returned. There are many examples of such protocols, each adopted to its domain; email (smtp, imap, pop3), www (http), telnet, snmp.

Where protocols are very much about the details of how communication is to be performed, Remote Procedure Call (RPC)/Remote Method Invocation (RMI), which is typically implemented on top of existing protocols, hides these details, and instead intends to let the developer focus on the objects and resources in the application by allowing remote and local objects to be manipulated in (almost) the same way. RPC/RMI is a codified way to construct generic APIs in a consistent way for business and domain specific applications (over a given wire protocol) ensuring interoperability and optionally language-native bindings.

While calling remote procedures is what RPC/RMI excels at, RPC/RMI can be limited in its capabilities of transferring objects. Strings, numbers and simple structures will work well, but if the RPC/RMI solution is constructed to be inter-operable between programming languages, it will be difficult to serialize complex objects in a way that is compatible with all the individual features of the involved languages.

Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) are two standards that try to go beyond RPC/RMI and provide truly distributed object architectures, by letting objects be defined in an Interface Definition Language (IDL), which then can be used to generate stubs for most popular programming languages. With a common definition of the objects in a framework where automatic generation of native bindings is well-supported, it is much easier to exchange full fledged objects over the network. On the other hand, relying heavily on a deeply integrated tooling solution, represents one type of coupling where one stands the risk of being limited by the options available in the development environment.

In the end, it depends on the application; for a service that is meant to be accessed by a web browser, the final presentation will be practically flat as it is laid out on the page, while many business applications will benefit from richer object representations and hierarchies.

After a development towards increasingly capable and sophisticated, but at the same time more complex and committee-driven specifications for exchanging information - culminating in CORBA primarily for business internals and later WS-*/SOAP for web services - we are now witnessing a reverse trend as popularity of RESTful ([129]) APIs is outgrowing that of SOAP, and CORBA is rarely mentioned anymore. In a way it has become full circle as REST - as a way of designing distributed systems [130] - is very much about using the HTTP protocol in an optimal way as is. Advocating to:

“maximize the use of the pre-existing, well-defined interface and other built-in capabilities provided by the chosen network protocol, and minimize the addition of new application-specific features on top of it.”[131]

With the most famous example being the world wide web itself. While REST is by no means a new concept, the ubiquity of the www has made the HTTP protocol the de-facto standard allowing it through corporate firewalls and home routers alike. For

6. Possible improvements

developers today, HTTP and REST is the best choice for reaching the largest audience when delivering a distributed application/service.

6.2.5. NoSQL - unstructured and simplified for scalability

The recent interest in NoSQL and other unstructured storage shows that relational databases are not always the best option for implementing data persistence. With very large amounts of data, in applications that are both read and write intensive, the overhead of the Atomicity, consistency, isolation, durability [132] (ACID) mechanisms offered by relational databases will make them slower compared to systems that do not perform as elaborate integrity checking.

Relational databases scales well within single-system setups, but not so easily by distributing the database across many physical systems. Here, NoSQL-based systems tends to fare better. NoSQL can also handle unstructured data, such as documents, multimedia and social media, efficiently. The trade-offs for NoSQL are on one hand, speed and scalability, and on the other hand, weaker data integrity, less precision and weaker support for complex data models.

6.2.6. Asynchronous and event driven

In the early 2000, the increasing usage of the internet raised concerns about the scalability of the then current web servers. This was postulated as the c10k problem; how to handle 10 000 concurrent connections [133]. Synchronous models with blocking I/O were blamed for not scaling well for large number of clients and it was suggested that asynchronous engines would fare much better. The ongoing shift from Apache towards nginx is a strong indicator of the implications of this realization.

Recently, a new document dubbed c10m as a namesake to its predecessor, was begun [134]. It summarizes what was done to solve the c10k problem and then proceeds to suggest what needs to be done for achieving 10 million transactions per second in the near future.

The framing is fresh in that it acknowledges that hardware is certainly capable of handling these kind of rates and goes on to suggest that it is the software, and in particular the popular OSes of today, that are the limiting factor, because they were not designed with these types of workloads in mind. The solution seems to be to go outside the kernel with the challenging tasks and run them in user space in much more specialized environments [135].

6.2.7. The high-level, resource-elastic and friction-less future

In “The New Kingmakers” [136] it is argued that FOSS have freed the developer from much of the bureaucracy in getting started with a software project, and that cloud technology has had a similar effect for hardware. Much of the earlier friction is gone, and as a result of being able to move faster with projects, developers have become empowered to take greater part in technology decisions; a potentially self-feeding effect.

6. Possible improvements

Developers tends to lean towards pragmatism and using the best tool for the job, rather than relying on technology that is “designed by committee” or enforced by company policies. Such a polyglot mindset can be the reason for the general increase in technology diversity and the recent popularity of unstructured storage and RESTful services. These are light-weight and straight-forward alternatives to more traditional solutions, characterized by a very low threshold for getting started using them and wide availability.

There are maybe no earth-shattering trends currently underway, but rather, as is often the case, slow working evolution and refinement of earlier ideas, continuously expanding our knowledge and repertoire of technology. The combined effect, however, can be profound. Less friction and increased productivity from - now more viable - high-level programming languages, promise a future where less code can do much more work by leveraging a vast machinery of distributed and carefully orchestrated computing resources.

From this we can extract a few guidelines for developing the (HPC) software systems of the future:

- Leverage the large body of available Free and Open Source Software as ready-made building blocks.
- Start out using high-level programming languages with high programmer productivity. Only turn to compiled languages for performance critical parts of the system when there are no other options.
- Design software systems to be loosely coupled to allow for scalability.

6.3. Middleware and distributed management

Middleware is functionality related to distributed computing captured in reusable packages. It forms a layer between applications and the operating system, across nodes and even sites. Typical tasks for middleware includes handling of message passing between processes and persisting information that is processed by the system.

Middleware is commonly used to create services for exposing application functionality in interfaces that can easily be consumed by client code. Organizing distributed applications as services has become known as SOA - Service Oriented Architecture. Services are encapsulation of software functionality between processes, representing modularity at a high abstraction level and can in this regard be considered one way of practicing code reuse (section 6.5.3). For increased automation and easier integration, one can make use of an Enterprise Service Bus to orchestrate SOA deployment.

For efficient exchange and storage of information it is wise to use a common data model and compatible serialization formats. Doing so also simplifies the implementation. Technology suitable for creating such a model and related middleware for HLT will be discussed next, as well as suitable specifications for distributed management.

6.3.1. Object-Relational Mapping

Object-Relational Mapping (ORM) is a programming technique for persisting data from object-oriented programming languages to relational databases. An Object-relational mapper is a software package that implements this conversion in a generic way, so that it can be applied to any object model to make it easier to implement persistent storage of objects to RDBS.

The advantages of ORM are the sum of the advantages of the two concepts it combines; Object Oriented Programming and Relational Database Management Systems. Object-oriented programming has since long been the predominate paradigm in software engineering. It promises better coding efficiency as software can be modeled closer to real-life objects, making it easier to grasp for humans. And it offers code encapsulation, so that the source code can be organized and subsequently engaged at the most suitable abstraction level for the task at hand. Relational Database Management Systems (RDBMS) are proven technology that have strong referential integrity and handles concurrency well. RDBMS have good overall performance, scaling fairly well, as well as elaborate security mechanisms.

Examples of ORMs are:

- Hibernate [137] for Java. It was one of the first of such mappers and is maybe also the most well-known.
- SQLAlchemy [138] for python is an ORM that includes a declarative way of defining objects, further simplifying the process of mapping objects to databases.

In addition, the fact that an ORM is a layer in between the business logic and permanent storage can bring benefits of its own, as it often means that the database back-end is made interchangeable. This makes it easier to scale up to more capable back-ends during the application life-time, from smaller memory based databases for prototyping/development/testing to full-blown server implementations for deployment. It also means that the properties of persistent storage - such as high-availability and scalability - can be manipulated through careful selection and configuration of database solution, mostly independently from the rest of the system.

The abstraction also makes the programs more vendor independent and less dependent on a specific database technology. On the other hand, there is a risk of becoming dependent on the mapper technology chosen.

6.3.2. Inter-process communication/remote API access

Twisted is an event driven network engine [139] that offers several ways of remote communication between processes. It ships with a lot of ready-made functionality that can be used as building blocks to quickly build distributed applications.

PerspectiveBroker is a Twisted module that offers easy copying of objects between processes and something described as “translucent access” to remote objects. The meaning being that a remote method can be called in a similar way as if it was used on a local object, except it is clearly marked as being remote [140].

6. Possible improvements

Asynchronous communication can also be achieved by deploying a message queue. AMQP Advanced Message Queuing Protocol is an open standardized protocol commonly used by several implementations.

ZeroMQ [141] was developed as a response to the gradually more committee driven standardization process of AMQP. It advertises itself as an intelligent transport layer, concerned mainly with transport and leaving the queuing to be handled by the developer. It has built-in set of sockets that makes it easy to construct a wide range of network topologies, such as publisher-subscriber and many more. It is extremely fast and efficient, supporting practically all relevant programming languages [141].

Lastly, a typical infrastructure for exchanging information between applications in enterprise is Enterprise Service Bus (ESB). Examples of implementations include WebSphere from IBM and Biztalk from Microsoft.

6.3.3. Data serialization

Serialization is the process of organizing a program's live data-structures or objects in such a way that they can be stored to disk or sent over the network for later to be easily restored in a semantically equivalent form. Serialization formats are often used between different systems and an important attribute is therefore interoperability.

Extensible Markup Language (XML) is maybe the most well-known human-readable format for interchange of information between applications and has a strong position in enterprise applications, in particular those that needs to integrate with a range of different systems. JavaScript Object Notation (JSON) is gaining a lot of traction following the wide-spread use of javascript, the programming language that it has been derived from. JSON objects can easily be transfered in and out of a javascript engine, and it is therefore popular in web services and REST-like implementations that communicates mainly with the web browser. JSON is considered lighter and less complex, while perhaps lacking in verification and built-in query tooling, when compared to XML.

Human-readable formats have the advantage of being easy to inspect visually, while binary formats can offer more efficient encoding of data. Defining binary formats in an IDL can help facilitate implementations in a variety of languages by automatically generating code. This is an approach used by Protocol Buffers from google [142] and Thrift [143] from Facebook.

There are also binary variants of JSON, such as BSON [144] that is used in mongodb [145] and MessagePack [146].

6.3.4. Service discovery

Service discovery is a general term for describing automatic detection of devices and services on a computer network. Two standards, Zeroconf [147] and Service Locator Protocol [148] (SLP) [148], are commonly used to implement service discovery software for operating systems.

Avahi [149], a Zeroconf implementation, has already been used in HLT for announcing the connection points of data sinks in the publisher-subscriber framework. This is where

6. Possible improvements

the event display is connected in order to show events directly from the data stream within HLT.

Unfortunately, the requirements were not sufficiently worked out, so the suggested solution and in particular its implementation did not fit well with the final requirements. See appendix D for more information.

The automation offered by service discovery; announcement of newly started services, continuously monitoring for new services and resolving of connection information of discovered services, fits well with the self-configuring capability known from autonomic computing, and is therefore natural to include as a part of an autonomic computing solution. However, it should be investigated if OpenSLP [150] is a better fit than Avahi. DMTF has already standardized on using SLP for service discovery of Web-Based Enterprise Management [151] (WBEM) services [152].

6.3.5. DMTF management standards and the WBEM stack

The technology mentioned so far, are basic building blocks that can be used to implement higher level designs of distributed systems. For distributed management systems, there already exists a set of complementary specifications, collectively called WBEM, that can be used to ensure development of compatible systems. WBEM itself builds on established web technologies and standards, like XML for serialization, HTTP for transport protocol and SLP for service discovery.

A core standard is the Common Information Model [153] (CIM), a model that describes the elements and the connections in-between in a managed environment. CIM is defined in Managed Object Format [154] (MOF), a IDL for describing managed objects, but is also distributed in XMI format in support of existing modeling tools, in particular UML modellers.

The model is then made available to clients with a CIMOM, the server in a WBEM stack, and providers are used to feed the information to the CIMOM from the clients. Two prominent implementations are SBLIM [155] and OpenPegasus [156]. The development and maintenance of the WBEM standards is overseen by the DMTF.

6.4. Developer tooling

The software developer's toolbox necessarily includes the most immediate tools for writing source code and finding mistakes, like an editor or IDE and debuggers. But tools for source code management, automation, testing and build systems are also important for a software engineering practitioner.

These can all be used independently by developers, but setting up common infrastructure saves a lot of work for the individual developer and opens up a whole new range of opportunities for improving the software engineering aspect of a project.

6.4.1. Source code management

HLT's source code repositories are all hosted in subversion, a modern incarnation of the traditional centralized Source Code Management (SCM) system. While centralized systems have for a long time been the only option, much of the innovation and momentum in source code management is now happening in the increasingly popular distributed version control systems (DVCS), git being perhaps the most prominent example. The many online collaboration services, offering hosted services (such as github [157] and bitbucket [158]) for these tools have helped propel this trend.

The main advantage of a DVCS, is that commits can be done locally, at ones own discretion, without having to be concerned about breaking code for others. Changes can be committed whenever it feels natural in smaller incremental steps, rather than in larger pieces, making it easier to keep a good history and narrowing the scope for when having to look for bugs.

With DVCS, each copy of the source tree is by its own right a full repository. Commits can be amended, reordered and duly prepared before changes are shared by pushing and pulling commits between repositories, often with an authoritative repository in between.

Cheap and powerful branching is another strong point of DVCS. One could say that DVCSs are more agile in this regard, and that they better support the work-flow of the individual developer (thereby enabling better practices), in addition to the general benefits traditional SCM systems brings to a project.

Most DVCS have functionality that allows them to work with a subversion repository remotely, while still allowing the user to operate the local copy as if it was distributed. It is therefore fully possible to have a gradual transition from a centralized repository towards a distributed one, by having users switch when ready.

6.4.2. Task automation

Shell scripts are perfect tools for automation of tedious and repetitive tasks. Sequences of commands can be safely stored in a file from where they can be run as many times as desired. Scripts are however not great for writing larger software systems, as has been mentioned earlier in section 5.2.4.

It is therefore common in automation systems, in particular for cluster-management and provisioning, to have the higher level logic written in a more capable programming language, while the lower layers are calling command-line tools and scripts.

Examples of this include Puppet, Saltstack, Rundeck and many more. These are to a varying degree transparent in their usage of underlying command-line tools. In some cases its completely hidden and a DSL - domain specific language, is used to describe tasks.

At the low-end of abstraction, but with high transparency is fabric. Fabric is a tool for streamlining and automating execution of tasks on the command-line [159]. It is a thin layer between Python and the shell that makes it easy to execute related blocks of commands either with the included fab command-line tool or from within other Python programs. When a task is described in Fabric, it captures both the step-by-step instructions for how a task is to be performed and the details of command-line options and

6. Possible improvements

arguments that are supplied to the tools in each step. In other words; the tool-set configuration of a project. Software like Fabric allows for a consistent way of running tasks across packages independent of differences in the underlying tools.

This can also be useful for continuous integration, where a single piece of glue code can be used on the integration server for all packages complying with the same specification. The continuous integration server then only have to deal with a unified and well-defined set of commands or tasks.

Within a fabric file, explanations and examples can be added with docstrings [160], which is a way of writing comments that flows well within the source code itself. Docstrings are well-understood by documentation creation software such as sphinx [161] and epydoc [162], and are included in the output of such tools inside generated API documentation. Tasks that normally would be run from traditional shell scripts, often having no documentation at all, can in this way easily be run - and documented - by using normal Python conventions and tools, in a solution that is quite close to the ideas in literal programming (section 6.5.5).

6.4.3. Build configuration

A proper build system is the first step towards predictable, controlled software development. In most software projects, a configuration or build system suitable for the application is included from the very beginning, since it is a great help for the developer to automate the build process as well as providing a recipe that can be used by others who wish to compile and use the software, for instance collaborating developers.

“make” has been the traditional build tool for a long time in the UNIX world. It is a low level tool in the sense that much of the functionality will have to be written by the developer. Autotools [163], has codified the usual tasks that you would have to provide yourself in make. It knows how to compile binaries, libraries and documentation.

In recent years, CMake [164] has become a popular alternative to Autotools due to having arguably simpler syntax and being less complex. Both Autotools and CMake are dependency aware and can configure the sources to be built for different platforms as opposed to pure make.

The relative simplicity of CMake may be attributed to the fact that it only needs one file for supporting all major platforms. From this file, build and project files are generated for existing build tools on the target platform. CMake can therefore be seen as a meta build system, where the responsibility to ensure that supported platforms have working build tool chains lies with the CMake community, rather than with the application developer [165].

When the infologger was adopted to the HLT cluster, CMake files were produced to replace the make-files included in the DATE package. The DATE package was made to be used with SLC [166], but the improved build process made the transition to Ubuntu easier by automatically choosing correct compiler options and allowing the configure process to look for correct paths of needed libraries. In other words, the build process had become more portable.

6. Possible improvements

The experience from the infologger inspired the effort to introduce CMake in the transport framework, which like many other packages in HLT, has for a long time only had hand-written make files.

With a properly configured build system, several versions of a library can be installed at the same time and dynamically linked binaries will still be able to select the correct library without having to consult environment variables. This can be achieved by setting the correct RPATH in the library of the binary as is explained in [167]. Such a change, together with proper packaging, would make the use of the modules system mostly redundant and would have reduced the amount of needed shell scripts, and thereby its draw-backs (see section 5.2.4), considerably.

6.4.4. Testing and continuous integration

The CMake suite also contains tools for testing (CTest) and packaging (CPack). When CMake [164] scripts were added to the infologger, CTest [164] functionality was enabled too, so that it would be possible to do acceptance testing of the infologger functionality when merging with new releases of the DATE package. The full cycle testing described in appendix F.3, means that all of the mechanics in the infologger is exercised in an automated way.

CTests are defined in the CMake files much in the same way as build targets for binaries or libraries are. The `ADD_TEST` command takes a binary or script, with optional command line arguments, run this command and looks for the exit code to see if the test failed or succeeded. CTest follows typical linux conventions where an exit code of zero means success and anything else is regarded as failure [168]. Failures are summarized after a test run has completed.

Unit tests can also be run this way and external tools can be integrated with CTest. For instance the Valgrind tool suite [169] for analyzing software problems such as memory leaks, and Gcov [170] for code coverage.

All of the test results can be gathered and shipped to a central build dashboard called CDash [171], where the trends over time can be observed. The CDash package can also function as a simple build server which the client nodes pull for information on when builds should be initiated, effectively performing continuous integration. Still, CDash is firstly a dash-board for build results. Other, much more powerful CI solutions exists out there, the most well-known being jenkins [172], but also cruisecontrol [173] and buildbot [174].

Continuous integration with unit tests and acceptance tests for the complete system, not only subsystems as done by the individual developer, vastly improves the quality assurance in a software project and has almost become a mandatory practice in software engineering.

Together with software analysis and metrics, continuous integration helps to increase the understanding of the application and the development process. It becomes easier to make good estimates and steer the development in the wanted direction.

With full test coverage and sophisticated acceptance testing all performed regularly by a continuous integration server, there is not much more that can be done to ensure that

6. Possible improvements

the application will work as expected. One might just as well release the application when a round of testing has completed. This is what has recently become known as continuous delivery [175].

6.4.5. Packaging and deployment

CPack supports creating source, DEB [176] and RPM [177] packages. Dependency information can optionally be included so that the package manager can automatically install dependent packages in advance. DEB packages were created for the infologger and could be directly injected into the locally maintained Ubuntu package repository.

This repository is also used for patched packages and for packages that are not available in the official repositories. Conflicts between local and official packages are resolved by pinning [178], a feature available in Advanced Package Tool [179] (APT), the package management tool used by Ubuntu.

By using existing tools that are built into the core of the distribution, all the benefits of using a proper package manager are ensured and all the available infrastructure that comes with it is available.

6.5. Software engineering practices

With common infrastructure and proper tooling in place, one can start to consider which software engineering practices can be useful for a given project. In the context of HLT, there are a few issues that immediately spring to mind. Most agile practices are meant to be used in teams working closely together on the same software product. Thus they might be difficult to apply in a distributed and diverse development group, as is the case for HLT. Secondly, many practices are considered to be most effective when used together as a set of coordinated practices. Making use of just a few of them might therefore be of little use. This section suggests a list of practices that are generally considered to have a positive effect, while still being relevant to HLT.

6.5.1. Code review/inspection

Code review or inspection is by many considered to be the most effective way of reducing bugs in code. Studies have shown that it can catch 60-90% of bugs even before the code has been executed [180]. It is most effective the first time one or more persons sits down to systematically read through code, and within the first hour of reading [181].

6.5.2. Iterative development

When software development became an industry of its own, it inherited the production flow from traditional manufacturing. Here the steps are typically separated into distinct phases, one feeding into another - design, implementation, testing, manufacturing - giving it the name of the visual analogy of a water-fall [182].

6. Possible improvements

The intangible nature of software makes it harder for stakeholders to reason about the desired end-product. It is difficult to articulate and communicate what a software product should look like and how it should behave. And so in the end, the agreed-upon requirements might not lead to the desired product.

Iterative development is a response to the traditional water-fall method of development. With smaller scope of each iteration and the possibility of having a positive feedback-loop, it tries to be more forgivable to changing requirements and let the process and software adopt as the common understanding of the system grows.

Estimation becomes easier and the quality of the estimates improves throughout the project. Each iteration should (according to agile methods) end with updated artifacts that can be used as input to metrics. Good metrics can be used to further improve estimates for the next iterations.

This way, iterative development can help alleviate some of the problems discussed earlier, for instance with big design up front (section 5.1.5) and in making metrics available earlier for effort estimation (section 5.6); metrics could be available already by the first iteration and not only at the end of the project.

6.5.3. Code reuse

The general virtues of code reuse, and its promise as the saviour of ever-increasing development costs (together with high level languages), have been touted at least since the seminal “Mythical Man-Month” [14]. Also the presentation of HPC productivity (section 5.6.3), suggests that code reuse should be a good approach for saving cost, but takes care to warn that a high percentage of reuse can be more difficult to achieve in HPC than other fields due to code optimizations made for the target hardware platform and the specific nature of the application itself [75].

Apart from the effort saved from not having to actually write the parts that are being reused, code reuse improves code quality as the code being reused normally has seen more testing and bug fixing than something created from scratch, basically leveraging earlier work.

Still, for all the alleged advantages, the vision of code reuse has never come to fruition in the way many had hoped for and many remain skeptical of its potential.

6.5.4. Test Driven Development

Test Driven Development [183] (TDD) encourages writing tests before implementing functionality. A test without any supporting code should fail and then just enough functionality to make the test pass must be written. When coupled with a practice of always looking for opportunities for refactoring, this approach is very effective in gradually and continuously improving the design of the software.

The small coding steps resulting from the usage of refactoring and TDD (see for instance Refactoring [184] and Refactoring to Patterns [185]) makes bugs visible early and since the scope of change is small, problems are easy to pin-point. The source code appears more transparent, comprehensible and well-tested. Furthermore, test code can

6. Possible improvements

serve as working examples for documentation, showing how to use the software at hand from a developer perspective.

Having a solid test suite builds confidence in the developer, making him or her feel safer about regressions being caught if they were to be introduced during development. While TDD has the potential to increase robustness of a software system, it relies on proper infrastructure and scaffolding, like unit testing software, build systems, mocking and continuous integration, to function effectively.

The general lack of unit tests and continuous integration in HLT (section 5.2.3) makes TDD very difficult to practice at the current state. A lot of work is required to get in place the needed preconditions for TDD.

6.5.5. Executable documentation and literal programming

Working software will always have higher priority than producing documentation. By making documentation part of the coding process, the hope is that it will encourage keeping documentation in sync with changes in the software.

Taking Python as an example, comments in source code are advised to follow a convention called docstrings [160]. These can be extended with reStructuredText [186] and doctests [187] to bring more structure to the documentation and to embed tests in the comments, respectively. When combined with automated test tools, these extensions can bring the development process closer to concepts like executable documentation and literate programming [188].

Executable documentation is not a well-established term in the literature, but can be seen mentioned in the context of executable specification [189] and Framework for Integrated Test (FIT) [190]. FIT and the more recent FitNesse [191] are collaborative tools for acceptance testing, where specifications are defined in tabular views that can easily be edited by non-programmers. While the input to FIT is Hyper Text Markup Language [192] (HTML), these files can be created in office applications - or wikis in the case of FitNesse - before being exported to the FIT software. These tools help contrast what an application should do and what it is actually doing. They use human language to a large extent, which greatly improves communication between stakeholders and developers. In addition to be the actual specification, these live documents, also capture much of what would normally be considered to be documentation. This is an additional motivational factor for using such tools.

Literate programming takes these ideas to its logical conclusion by putting the human language at the center of attention. The source text of a literate program is written very much like prose, but with the added feature of macros that can be used to hide complexity and interweave blocks of traditional programming language statements. Also here, documentation and executables are extracted from the same sources. Despite having been around for a while, literate programming has never really caught on. However, ideas from literate programming seem to have inspired more recent efforts in improving the specification/documentation writing process, such as the Behaviour Driven Development [193] (BDD) methodology [193].

BDD builds on the experience from TDD, but moves the focus away from strictly

thinking about tests towards talking about what behavior software should have. The motivation is also here to bridge the communication gap between the stakeholders and developers. Much of the development of tools for BDD is happening in dynamic languages, with cucumber [194] being a prominent example.

6.5.6. Collective Code Ownership

Common infrastructure makes it possible to practice collective code ownership as it is known from Extreme Programming (XP) [195]. Collective code ownership is often mistaken for non-ownership, which would normally lower the quality of the software, but the emphasis should be on the “collective ownership” of a system or subsystem. However, for collective code ownership to work well, several other practices from extreme programming must be in place, such as extensive unit test suites, continuous integration and strong coding guidelines [196].

The benefit of practicing collective code ownership, is that developers typically feel a greater responsibility for the software and have a greater stake in its success. They are empowered to make changes that accommodates their own design, as long as they make sure that no unit tests are broken. The burden of communication overhead and bureaucracy is therefore reduced to a minimum. Instead creativity and drive in the development process is stimulated, while still allowing for the overall design to improve.

6.5.7. Model driven engineering (code generation)

In a software project, working out the specific details of the target domain and finding the relations between the contained elements is often the most difficult and time consuming part, while the implementation afterwards can be straight-forward albeit still laborious.

Given that a good model is found, and the steps for producing a representation of the model in code are trivial, then the particular case at hand lends itself well to model driven engineering, where if sufficient tooling is in place, the defined models can be transformed directly into code.

6.6. Monitoring, logging and debugging

It is important to clarify in the planning stage which components already exist and can easily be reused in the implementation, and which pieces are missing and will have to be built from scratch. When modifications are needed in order to adopt software to the needs of a project, those efforts must be weighted against the effort of building something new.

The temptation to create something new will often be strong, but one should keep in mind that:

- In-house developed software often take longer to finalize than expected; there is more work than what was anticipated.
- Maintenance work can be shared by more hands in a bigger collaboration.

6. Possible improvements

- Packages initially deemed unfit will improve over time, maybe faster than expected, and can make the in-house effort redundant.
- A “greenfield” effort (i.e. starting from scratch) might be much greater than participating, and contributing, the missing parts to a project that currently does not satisfy the requirements.

In short, there are several reasons why re-evaluation of the software needs in a project can be beneficial at a later stage of the development. If needs are found that are not currently being served optimally by HLT software, alternatives should be identified and installed to complement existing functionality. In particular when considering that HLT has become of age and many new projects have sprung up since its inception.

Section 7.2 in the next chapter will evaluate existing packages that can be put to use in HLT. The rest of this section will discuss improvements for systems that are already deployed and areas where new development might be needed.

6.6.1. Live monitoring and logging

HLT employs several systems for live monitoring. For the physics application itself, there is the infologger, that is described in 4.4.4.2 and 5.5.2. For cluster management in general there is the in-house developed SysMES package and ESMP. ESMP provides in addition chain control and graphs of resource usage on the compute nodes. All these distributed applications send their data to a central database in their own application-specific streams. This makes it difficult to correlate log events in timeseries data.

There are a few simple measures that could be made to improve upon the logging in HLT:

- Centralized logging is supported in the rsyslog daemon used in the cluster and can be enabled for the system logs of all the nodes for easier access and correlation. The centralized database can be setup with a web interface for showing the live feed and for querying of past log messages. This would add system logs to the existing centralized information sources.
- The in-house developed systems can also be setup to log to a centralized rsyslog in addition to their main service. This way, one could have both system and applications logs merged into one unified stream.
- SNMP information from the HLT hardware can also be gathered centrally with a tool such as cacti in addition to the monitors deployed by SysMES.

Using a centralized and unified logging system with all the available sources (syslog [82], snmp [197], infologger) is achievable with existing tools. There is, however, a tool missing that can help correlate and present the wealth of information in a prompt and intuitive way.

- An alternative here could for instance be Splunk [198], which advertises filtering and correlation of information from multiple (log and monitoring) sources as one of its strengths.

6. Possible improvements

- For monitoring there are several solutions available that could supplement already deployed systems:
 - Nagios [199]: industry standard IT infrastructure monitoring.
 - Zenoss [200]: monitoring and management of IT infrastructure.
 - Ganglia [201]: scalable, distributed monitoring system for high-performance computing.
 - Lemon [202]: server/client based monitoring system.
 - munin: [203]: The light-weight alternative for smaller installations.

The main sources for data-mining would be the logbook and the infologger database, but also SysMES and ESMP stores interesting data, as well as the bug tracking system (Savannah both HLT and ALICE). In the bug tracking system, issues are reported manually by experts, rather than by automatically by software. The logbook is in this regard a combination, containing both machine created information and reports added by shift personnel.

For better data analysis and improved pattern matching one should consider deploying structured log messages in addition to centralized logging. Improving support for structured log messages has seen renewed interest lately ([204], [205]). The technical specifications needed are already in place [82, 206] and the logging community is pushing to implement support in the standard logging software found in today's Linux distributions.

In light of these recent developments, one should consider moving away from infologger and rather use a standard logging system that offers centralized logging and the option to send structured log events. rsyslog, which is currently in use should in the near future meet both of these requirements.

6.6.2. Metrics gathering and visualization

Related to monitoring and sophisticated logging is the gathering and visualization of metrics, both business and hardware metrics. Visualization is also important for monitoring.

There are several things here that could be improved in HLT. There are ready-made packages that can be installed: StatsD [207] for business metrics, collectd [208] for hardware metrics. The metrics can be fed to graphite [209] or rrdtool [210] for time series storage and real-time visualization. Also past-time visualization is provided.

6.6.3. Debugging large and complex systems

The live monitoring just mentioned is of course very useful, but must be considered a precondition for any meaningful operation of such a system. The transport framework do have the possibility to do event accounting, meaning one can get a detailed dump of how the events are traversing the chain of components.

6. Possible improvements

Except for event accounting, the tools available for debugging include the command-line and the normal UNIX tools, such as `gdb`, `strace` and similar. Connecting to a problematic process with `gdb` and stepping through a malfunctioning routine can be extremely helpful, but when there are thousands of processes spread over a couple of hundred nodes, it will quickly become difficult to follow the distributed program flow.

The standard tools don't scale well, but there exists some commercial tools, that claims to do so, such as TotalView [211]. This package can be used in scalable debugging, meaning it handles parallelism in distributed processes as well as in threads. Packages such as these seem to be tuned for usage with Message Passing Interface (MPI) though, which is not used by HLT.

6.7. Summary

Reducing the amount of mundane and repetitive work has been a great benefit from the automation enabled by software innovation. Initiatives like autonomic computing are pushing automation and autonomous systems to handle the growing complexity in large computer systems.

With autonomic computing, current trends and established practices in software engineering as guidance, this chapter has presented suggestions for improvements that can be beneficial for projects like the ALICE HLT.

Much of what is said about methodologies and technology in this chapter is generally applicable to software projects. This can be taken as an indication that software development in HLT, and maybe HEP in general, could have much to gain by making use of established practices from the software industry:

- Increased use of agile practices, in particular those related to test-driven development, such as unit testing, mock objects [212] and continuous integration would have made it more feasible to test sub-systems thoroughly in isolation, improving the quality and robustness of the code.
- Collective code ownership could have helped created an environment in HLT which could attract new talent while also keeping developers that was already involved.
- Increasing the use of supporting software artifacts, would improve maintainability, so that one could more easily build on top of the existing code base throughout the lifetime of the experiment, but also so that the code can more easily be reused in other projects.
- Making use of some sort of executable documentation as described in sections 6.4.2 and 6.5.5 would have significantly improved the quality and relevance of documentation for cluster administration. This in turn would have helped the continuity in the project, avoiding much of the redundant (re)work being done for infrastructure and cluster administration.

6. Possible improvements

In moving towards a more formal process in stages, a first step can be establishing guidelines for all new development to follow, to make sure one moves in a coherent direction. Next, one can establish design documents that describes in more detail how things are working and how things should be done.

To the list in the summary of current trends, one can add guidelines from the developer tooling and software engineering practices subsections:

- A common repository for source code should be used.
- Prepare continuous build- and integration servers for use by the project.
- Use service discovery and strings instead of hard coded IP addresses and port numbers.

7. Design and implementation

Earlier chapters have addressed HPC in HEP/HLT in successively smaller scope, finally narrowing it down to cluster management software in this chapter. The composed cluster architecture enabled by COTS, has its inherent challenges in that the risk of malfunction - in hardware and software - grows with the number of components in a system, and thereby, the size of compute clusters. Although the situation is improving, hard problems remains that are yet to be fully resolved, while clusters continue to grow in size [37]. Cluster management software is here to help us cope with these challenges.

In this chapter, design concepts and technology that can improve the autonomic aspects of HLT management software, will be evaluated and verified or falsified in prototypes. The methods and techniques that have been suggested in earlier chapters for improving the development process, will be sought used to aid the development and for creating metrics that can be compared to those presented earlier. Making use of the suggestions made, will provide an opportunity for an evaluation of those said suggestions as well.

7.1. General requirements

In a system like HLT, where so many different software packages must work together, one of the most noticeable omissions from the original design documents, is a common and unified infrastructure for communication between applications. While the data-transport framework is the way of transferring data between analysis components, and the task-manager is used for setting up and controlling an analysis chain, there is no common command and control system that can be used by all applications.

Most applications therefore have their own way of exchanging messages in HLT and common functionality is bound to be implemented repeatedly. Opportunities for code reuse and building more powerful, cooperating applications, are lost. Not leveraging reuse in software building leads to higher maintenance cost and having to rely on more people for knowing all the systems in day to day operation. The general benefits of code reuse are briefly discussed in 6.5.3.

The lack of frameworks or middle-ware that helps making it easier to write distributed cluster software is a general one. More specific to clusters like HLT are the benefits that could come from better resource management and automated solution for gathering inventory information. In addition to functional requirements, it is also the intention to verify or falsify some of the suggestions made in earlier chapters. This places additional restrictions on the implementation:

- Make use of high-level dynamic language. See if recent advances have made dynamic languages more competitive also in compute-intensive environments where

7. Design and implementation

performance earlier has been a prohibiting factor. Check the effect of applying optimizations like Just-in-time compilation (JIT) and Software Transactional Memory (STM) in PyPy [213] or adding partly static typing like made possible in Cython [214].

- Research object-oriented remote invocation mechanisms to be used in the implementation, with the premise that such mechanisms will make it easier to implement natural language queries for management scripting/coding in line with the fundamental user properties of autonomic computing (section 6.1).

7.2. Existing software

The best possible case of reuse is when an existing software package can be deployed as is, without modifications. Software can gain some flexibility by being made configurable, and sometimes, tweaking configuration options will be enough for integration in the target environment. It can, however, be difficult to find a package that matches exactly with the requirements of a project, and it is therefore important in the planning stage to take sufficient time to investigate the available options.

As the WBEM technologies are made for management of distributed systems, software packages built on these are of high relevance. Amongst the mature and active implementations there are both commercial ([215]) and open source ([156], [155]) offerings. Most implementations have been written in C++ or Java, but there is also a python client library and provider interface for working with WBEM infrastructure, called PyWBEM [216].

There are also management solutions which are not based on WBEM technology, with a stronger focus on clusters. BrightCluster Manager [217] is a commercial and feature-full representative of these, while the open source Pacemaker [218] together with Corosync [219] can be used to provide highly available services. The Corosync Cluster Engine, a core infrastructure for highly available clusters, provides the messaging layer, while Pacemaker [218] is a scalable High-Availability cluster resource manager layered on top of Corosync. This type of high availability software packages are very useful for system services such as DNS, DHCP, NTP and file servers. For the HLT application, these tools might not be very useful by themselves, but they could be part of the solution to bring high-availability to the HLT chain. A similar solution to the Pacemaker and corosync combination, Linux-HA with heartbeat [70], was successfully used to add high availability to the infologger setup in HLT (see 4.4.4.2).

Similar to the cluster managers are the workload managers or task queues. The Simple Linux Utility for Resource Management [220] (SLURM) is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters [220]. It is used by many of the most performant computer systems in the world.

The difference between these solutions and the concepts they embody is rarely clear-cut. The functionality needed in a cluster can be supported by a single solution or a combination of several. Some, like the Bright Cluster Manager combines most of the

7. Design and implementation

aspects in one package, while others can be used both stand-alone and in combination with others. One will often have to weigh “turn-key” and closed, with flexibility and maintenance when evaluating their suitability.

For secondary purposes, but also made for scaling management of distributed infrastructure, there are good solutions for advanced logging, several solutions for monitoring and some for debugging, as can be seen listed in section 6.6.

A bus-like infrastructure for information exchange, including functionality for keeping track of active nodes and services, would go a long way in being a solution for making construction of communicating applications easier. A lot of the needed functionality for this can be found in the packages just listed, and ideally more time should have been spent on evaluating them for getting a full understanding of how they work and what they can provide in the context of HLT.

However, having the privilege to observe the HLT project throughout its life-cycle and taking the requirements defined into consideration, it becomes evident that it is infrastructure similar to WBEM that is sought, but not with the confines of distributed management only. Rather the information exchange should be more generic, including most kinds, with the possible exception of high throughput data flows.

When it comes to commercial solutions, they are inherently difficult in that they are usually closed. Study of the source code and contribution in an academic context can be problematic. The goal to investigate the feasibility of using dynamic languages in the core of the infrastructure, also disqualifies the reviewed solutions.

Finally, the packages just discussed are to a greater extent polished products with a more or less well-defined application domain, while it would also be possible to choose from integrated infrastructure like ESB or more low-level technologies like ZeroMQ, in deciding which building blocks should be used and which granularity is suitable for an implementation.

7.3. Cluster API

Several applications and software packages are used to operate and manage the HLT cluster. Some are in-house developed, some are developed at CERN, while others are standard packages installed along the operating system. These applications together with their configuration is what makes up the "HLT application", meaning how hardware and software are put together - applied - to perform a certain task.

Many of these applications will need access to the same information and they would benefit from being able to make use of functionality already implemented elsewhere. This requires a type of shared means of communication that is rarely employed amongst cluster applications today, despite the existence of the fundamental technology needed, as can be seen in service oriented, REST and similar architectures.

In HLT, there have been no common guidelines or standardized ways of implementing inter-application communication. It is only applications with the explicit purpose of transferring data between systems (ALICE Off-line framework (AliRoot) interfaces) or software that were made to work together from the beginning (ie. TaskManager and

7. *Design and implementation*

pub-sub framework) that exchange information.

7.3.1. Use cases and requirements

Use cases are presented for the core functionality as well as for typical client services in order to build a complete picture of what a solution should look like. Client service examples are largely based on existing HLT software. The use cases used to derive the core requirements are shown in figure 7.1, while use cases that are relevant for cluster management and in particular the HLT cluster can be found in appendix C.

7. Design and implementation

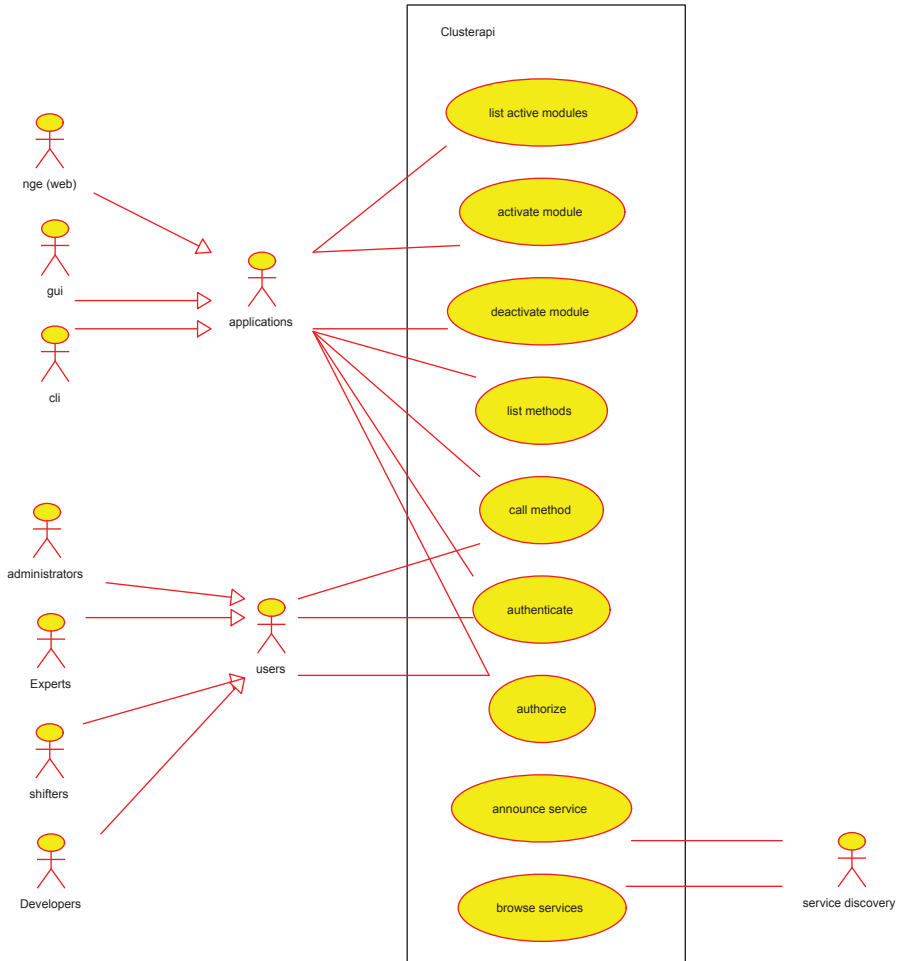


Figure 7.1.: Core use cases for clusterapi.

The discussed use cases are mainly concerned with facilitating access to existing functionality in distributed software services, and as such, the core application must support the operations that will help fulfill this goal. These are summarized in the following functional requirements:

- It must provide the mechanism for making existing code definitions available over a common API protocol to remote and local client code, with little to no modification of the original sources.

7. Design and implementation

- It must allow client code to browse for methods made available as described in the point above, without the client having to know the location - be it local or remote - of the method.
- Finally it must provide the mechanics needed to let announced methods be called by browsing client code, and return values as needed.

Non-functional requirements are derived from suggestions and guidelines in earlier chapters:

- There should be support for the most commonly used programming languages in HLT (see table A.3): C++, Java, Python and simple programs for command-line usage.
- Functionality should be possible to make available in Graphical User Interface (GUI), Command Line Interface (CLI) and web applications.

7.3.2. Design

Announcement and browsing of peer services can be implemented by using existing solutions for service discovery, such as Avahi or SLP. Service discovery hides the details of IP addresses and port numbers, and instead allows the programmer to address services by human readable strings. Daemons and developer libraries for service discovery are available for all major Linux distributions.

For automating announcement and browsing of methods, one can use reflection and introspection. Reflection is the act of dynamically discovering the internals of a program while it is being executed - and acting upon the information found [221]. It can therefore be used to construct a service API on-the-fly by scanning live code, for instance when a service is dynamically loaded. Introspection is similar to reflection, but here a distinction is made between the process that goes on inside the service to create an external representation of its internals and the peer discovery between services for learning about remote methods - the latter denoting introspection. In the long-term, it is also the goal to largely automate the presentation and interaction in a web interface based on introspection of the generated API's.

The point of a common communication framework is of course that it should be used by all kinds of applications, but in HLT, as in many other clusters, there are use cases more concerned with the data flow in the main application, while the rest focus on the control and information presentation part. It would therefore be beneficial to divide the use cases vertically, resulting in a layered architecture, where both layers extend throughout a cluster, but are used for different purposes.

The command and presentation part - from here on called `clusterapi` - will be the core of the developed prototype and the scope will be limited to this layer from here on, while the data flow part (`clusterbus`?) should be seen as a possible future extension. It should nevertheless be mentioned that such an extension would have to handle high data rates and volumes in advanced routing configurations. Message Queues, such as ZeroMQ [222]

7. Design and implementation

or RabbitMQ [223], might be good candidate technology for an implementation. Some use cases for such a system have been collected and are listed in appendix C.9.

Taking these preliminary technology choices into consideration with the overall requirements, one can start to flesh out the design:

- Clusterapi will be implemented as a peer-to-peer network of typical UNIX/Linux daemons whose main purpose is to maintain a shared up-to-date lookup table of all clusterapi methods in a cluster.
- Methods will be organized in name-spaces according to the service they belong to in order to avoid name-collisions and to make it easier to navigate.
- In order to achieve this, the daemon will first have to be able to learn about all other clusterapi daemons running in the cluster. Service discovery will be used to implement discovery of clusterapi instances between nodes:
 1. When a node comes up, clusterapi announces its presence via service discovery SLP and the node queries for already running daemons. Any already running daemon will be notified of the new node and references will be kept for those nodes that have been discovered.
 2. Likewise, when a node is stopped or becomes unavailable, it will be announced to remote peers so that references can be updated accordingly.
- Clusterapi will provide methods for listing available services in a cluster, and methods for starting and stopping them. The pattern is very similar as for node/daemon discovery:
 1. Services that are started will have their methods made dynamically available in the local clusterapi instance by the help of reflection. When all methods of a service are ready, the presence of the service will be communicated to all remote instances, which will then perform introspection to discover the methods and to keep a references for the future.
 2. Likewise, when stopping a service, the service with its methods will be removed and an announcement to that effect will be distributed to remote instances which will update their local references accordingly.

Dynamic, high-level languages, lend themselves well to reflection. In the case of the clusterapi prototype, twisted (Python) will be used to create a XML-RPC API. A language agnostic service API implemented in XML-RPC, can easily be consumed by client programs in virtually any programming language. A clusterapi implementation would have to implement these protocols for discovery and introspection in order to be compatible.

7.3.3. Core concepts

Conceptually, clusterapi can be seen as a directory service of distributed methods that are of common interest to software in a given cluster. It should enable creation of

7. Design and implementation

unified, cluster-wide and language-agnostic APIs, that makes the functionality of software deployed in a cluster easily available to client code and clusterapi enabled packages.

Since a system like this would first and foremost benefit developers in simplifying how to build cooperating cluster applications, it can be useful to try to envision how code would look like in the finalized system. Consider then the following, where node state information is used to decide whether or not to start a virtual machine on a node, as a short demonstration for how such an interface could look like for HLT:

Algorithm 7.1 Demonstration code for envisioned clusterapi.

```
1 from clusterapi import inventorydb , sysmes
2
3 for node in inventorydb.getAllNodes():
4     if not node.state == PARTICIPATES_IN_RUN:
5         sysmes.startVirtualMachine(node)
```

From this short snippet one can see that the majority of the code is made out of pre-defined methods. By being conscious about adding new methods as they are discovered to be of general use, a vocabulary will be established over time, that will also help define the application domain of the cluster.

Clusterapi defines the high-level protocols and technologies that should be used for announcement and discovery of nodes and the respective methods of the loaded modules on a node. At the core, the clusterapi design is about these concepts:

Unified access There is one shared codified way of accessing remote functionality and make functionality available for remote consumption. Unified access here also includes the abstraction of the location of distributed - and for the cases it applies - centralized information.

Defining the application domain Defining a vocabulary for the application domain in a system-wide API is a constructive way of finding the limits of a system and what it should be able to do. Doing so shifts the focus from the underlying technology towards the application domain, bringing the programming environment closer to the domain experts.

Sharing of functionality By defining and formalizing an api for the application domain, it becomes easier to reuse code across applications. Instead of duplicating code that deals with the cluster in general over several applications, one can implement functionality to be shared in the clusterapi and use it from the application. This might require a change of mindset, as one would have to think about which part of the functionality that is about to implemented, can be useful to other applications.

Discoverable Definition of the domain and unified access should be supported by introspective functionality, such as service discovery for discovering all clusterapi enabled machines and introspection of the modules loaded in the respective clusterapi instances to discover all available methods. With this information in hand, reflection

7. Design and implementation

can be used to build the presentation layer automatically, acting as a user-friendly mirror of the information available in the system. This is in line with the data centric view of autonomic computing.

Loosely coupled With a loosely coupled architecture, there are no strong requirements for programming language, operating system or technology used by the client application internally, which means the possibility of vendor lock-in is reduced.

Language agnostic It is beneficial if the api is language agnostic, so that developers can more freely choose which tool to use for a given job. This would make the functionality available to many types of users. System administrators could access the api from shell scripts. Physics applications could use C++ and management applications could be accessing the API via languages such as python and Java.

Flexible usage Clusterapi should not impose restrictions, architectural or otherwise, in the ability of client code to be used stand-alone. It should rather encourage a structure that allows client programs to be flexibly used, for instance on the command-line also outside a clusterapi context. This way modules can be useful in more settings and development/testing is simplified. One should easily be able to write a small script that can also be consumed by the bigger infrastructure.

7.3.4. Implementation

The remainder of this section will describe the current implementation in more detail.

7.3.4.1. Process management

Process management is needed for starting and stopping services in clusterapi. It is typical to run services as daemons, with the operation controls not visible to the user in a GUI, but running in the background, always being active when the host system is powered.

There are several solutions available for daemonization and related tasks that can be used to support this functionality in clusterapi. From researching alternatives, supervisor [224] was found to have suitable process management and to already provide a XML-RPC interface for remote access. In addition there is also a mechanism for announcing process state changes to processes that are marked - in the configuration file - as event listeners, making it easier to know when to trigger the announcement of a service.

Supervisor starts up the processes and takes care of all low-level process management, like redirection of standard input/output, error handling and logging. It holds a direct reference to the child processes, so it will always know if the process is running, it does not have to infer the state from the existence of a Process Identifier (pid) file.

Using supervisor leads to an implementation where clusterapi itself is being managed by supervisor when starting up, making supervisor a hard dependency. The two will always have to be started together as a lower requirement for providing the full clusterapi functionality.

7. Design and implementation

This should be fine, though, as one should see the two as one instance having cooperating components, each with separate concerns; supervisor for low-level management of processes and the core clusterapi for providing the abstraction of locality for distributed services. This separation is also advantageous for reliability, as non-clusterapi functionality of services managed by supervisor would not be affected by potential problems in clusterapi itself. Instead of a full service failure, it would rather be sufficient to manually recover clusterapi functionality.

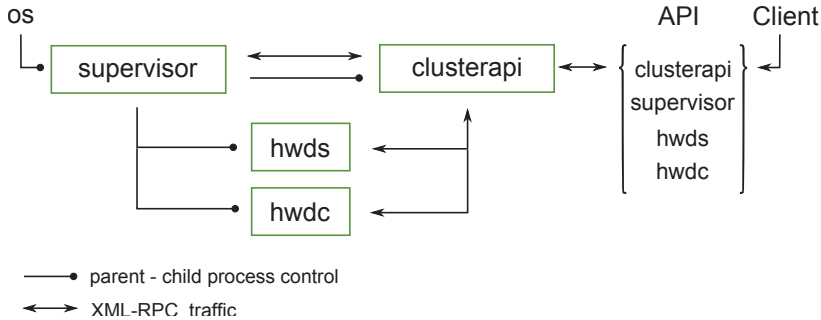


Figure 7.2.: Overview of process control and XML-RPC traffic in clusterapi. In this case, two services have been started; hwds and hwdc. Their methods have been exposed through the clusterapi and supervisor manage the processes.

7.3.4.2. Local service reflection

As part of this prototype, the functionality of supervisor has been augmented in two ways; with a XML-RPC extension that allows connection information of services defined in the supervisor configuration file to be looked up by clusterapi and by loading the clusterapi as a child-process under the event-listener configuration stanza, so that events emitted by supervisor are propagated to clusterapi.

This means clusterapi will be notified when a service is started and will be able to look up the service's connection information through a XML-RPC call (`getXmlrpcUrl`) if it has been defined in the supervisor configuration file under the environment option of the service. Events are communicated with a text protocol over standard input/output.

When having been started by supervisor, clusterapi will first query the supervisor XML-RPC interface of its (through known port on localhost) methods and add all of them to its own sub-handler under the namespace "supervisor". At this stage the clusterapi is ready for use, with all supervisor methods being exposed by clusterapi, the clusterapi instance being announced to peers and peers having been browsed.

From here on, processes can be started by clusterapi. This will trigger an event in supervisor when the process is running, that is propagated to clusterapi. The event contains the process name of the service which can be used with the `getXmlrpcUrl` call that will return the connection information needed to talk to the service.

7. Design and implementation

If there is a valid response, the service will be inspected and its methods will be added under a new sub-handler in the clusterapi. If a process is stopped, the corresponding sub-handler will be removed, including its methods. This dynamic addition and removal of local clusterapi compatible service methods is depicted in figure 7.3.

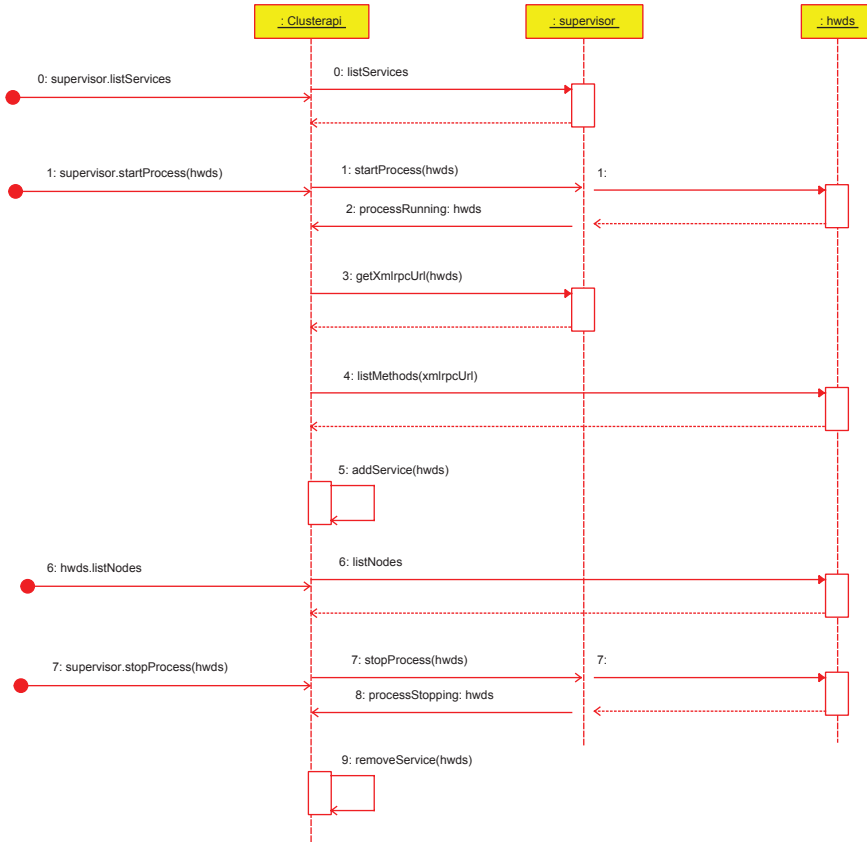


Figure 7.3.: Supervisor and clusterapi interaction. Here shown when starting a third process, calling a method on the started service and finally stopping the same process.

7.3.4.3. Remote service discovery

Since local and remote services both are made available over a XML-RPC interface, once a node with a clusterapi instance has been found, the same mechanics are used with remote hosts as for local service introspection. It is only the clusterapi instance itself

7. Design and implementation

that is being exposed through service discovery in the traditional sense, the presence of clusterapi-enabled sub-services are mediated through clusterapi itself.

The current service discovery implementation used in clusterapi is Avahi [149]. The experience from using it with the analysis chain in the HLT cluster suggests that it might not be well suited for installations with a large number of nodes (see 6.3.4). Implementations of SLP - or just using plain DNS service records - would probably be a more scalable choice. An example of a startup sequence with two hosts can be seen in figure 7.4.

7. Design and implementation

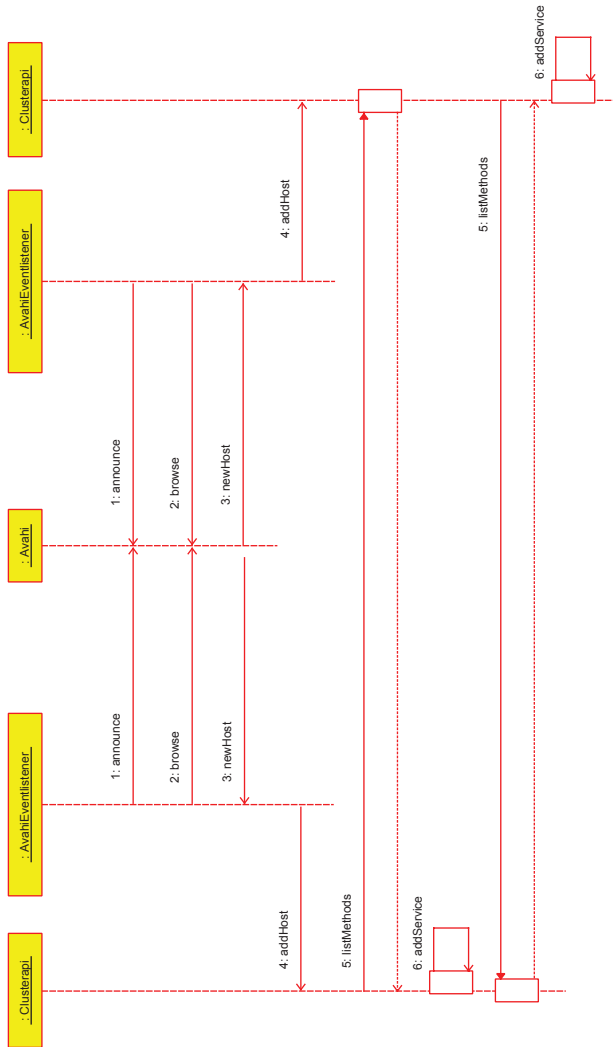


Figure 7.4.: Sequence diagram of two clusterapi instances starting up on two hosts, using service discovery. They act as peers, having symmetric sequences. The announcement and browse steps sets everything up so that the following steps are driven by events.

7.3.4.4. Eventloop

Clusterapi uses the eventloop of the event-driven twisted framework. The loop is shared between the XML-RPC service, the supervisor event listener and the service discovery listener. The two listeners will call methods on the service class when receiving events from external processes - the supervisor and the Avahi daemon. This is shown in figure 7.5 and a diagram of the involved classes can be seen in 7.6.

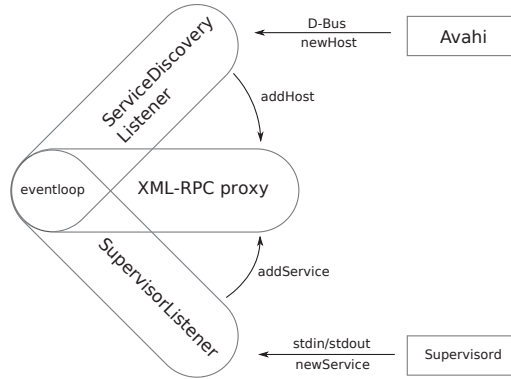


Figure 7.5.: Service discovery and the supervisor listener shares the event loop with the XML-RPC proxy. Service discovery information is communicated over a D-Bus interface, while supervisor communicates over a standard input/output line-based protocol.

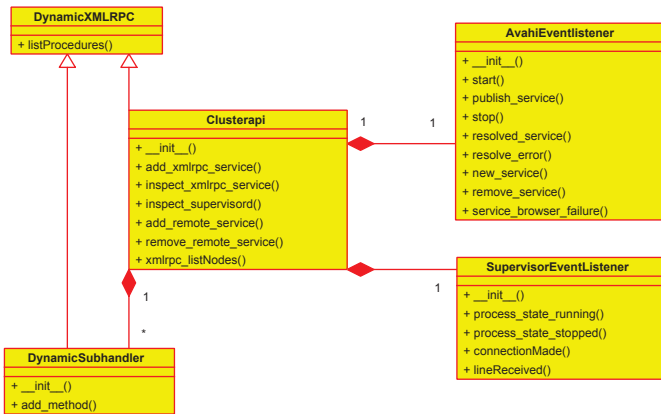


Figure 7.6.: Class diagram of clusterapi.

7.3.4.5. **Method dispatch**

When having established clusterapi as a single point for method dispatch, it becomes a point where optimizations can be made on behalf of the user. If a method happens to be available both locally and remotely for a clusterapi instance, it should ideally call the local one, assuming that the “closest” one is the most efficient. It will eventually be an exercise in weighing bandwidth usage when querying over the network to resource usage for storing/caching locally. The ability to choose the method that should yield the best performance can be sought built into the method dispatch mechanism.

One way of doing this would be to have a presorted list of hosts per method ordered by response time and always call immediately the one with the currently best rating, assuming that it is more important for the overall response time to save time in deciding which one will be the fastest now, than the potential gain in selecting the actual optimal method location.

The response time for the host is recorded after each call and the list reordered if needed so that its always the host with the lowest response time that sits at the top of the stack. The previous response time is seen as a good enough indicator for where a method call should be made and it will adjust itself continuously over time, accommodating a form of load-balancing.

When starting afresh, such a list should be pre-populated with values that would initially favor local methods, or one could use the response time of pinging the remote hosts - for instance gathered during service discovery - as initial seed values.

A proposal for a suitable data structure is shown in listing 7.2 and a sequence diagram of two consecutive calls are depicted in figure 7.7.

Algorithm 7.2 Data structure in JSON representation for method dispatch.

1. Method name, including module name-space, is used as a key for a dictionary lookup.
2. The value is a list of tuples, where each tuple contains the location (hostname) and a weight.
3. The weight corresponds to an approximation of the expected latency based on earlier recorded values. It is in other words an heuristic, not necessarily entirely accurate for the given moment, but an estimate considered good enough.
4. Per default, the method with the lowest weight is chosen (its at the top of the stack).
5. The method is executed and the time it takes is recorded for the location and stored in the sorted dictionary.
6. The dictionary is reordered if needed.

```
{
  'method1.service1': [
    (10, 'localhost'),
    (20, '1.some.host.org'),
    (27, '2.some.host.org'),
  ],
  'method2.service1': [
    (10, 'localhost')
  ]
}
```

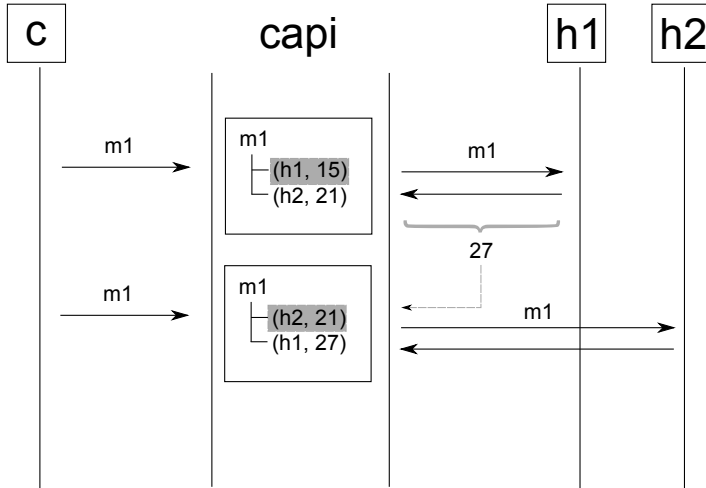


Figure 7.7.: Method dispatch example for two consecutive method calls. The increased response time for host 1 causes a reordering so that the next call to method 1 goes to host 2.

7.3.5. Core API

The core methods of clusterapi/supervisor:

Clusterapi.add_remote_capi Called when a clusterapi instance has been discovered by Avahi. Takes hostname, address and port as arguments and add these to an internal dictionary. Then calls “add_xmlrpc_service” with the service information.

Clusterapi.remove_remote_capi Called when a clusterapi instance has been discovered to not be active anymore. Removes references to the remote clusterapi instance from the internal dictionary.

Clusterapi.add_xmlrpc_service Called when supervisor announces a new XML-RPC service. If the new service has connection information from supervisor, a connection will be made, the service will be inspected and its methods added to clusterapi.

Clusterapi.remove_xmlrpc_service Called when a XML-RPC service goes away. The service will be removed from its sub-handler.

Clusterapi.listMethods List all methods available in a clusterapi service.

Clusterapi.listNodes List all hosts known to the clusterapi instance.

supervisor.getAllProcessInfo Get information of all the processes supervisor manage.

7. Design and implementation

supervisor.getProcessInfo Get information about a particular process.

supervisor.startProcess Start a process.

supervisor.stopProcess Stop a process.

7.3.6. Future improvements

In developing the current prototype into more mature implementation, there are several possible extensions that would increase the versatility of clusterapi and broaden its appeal, if added.

At first there is security that has been deliberately left out of the prototype scope under the assumption that most clusters are operated in a protected environment with access only allowed through protected gateways, and therefore not being considered an important enough aspect to be demonstrated in a prototype.

However, with better authentication and access control, it would be possible to allow greater interaction from a broader audience without fear of misuse or someone breaking the system - deliberately or not. For best results, such mechanisms should probably be implemented at the core of clusterapi. Kerberos would be a candidate for distributed authentication with its single-sign-on functionality that could be integrated in clusterapi.

Secondly, there is potentially a lot of functionality overlap between clusters. Certain operations related to the underlying technologies used are independent of the main cluster application. With a division in core functionality and sub-services, there will still be a lot of service functionality that can be shared across clusters. Thus, such functionality could be shared as standardized libraries bundled with the clusterapi. Good examples would for instance be services that integrate closely with system services, such as inventory gathering and user management, services that typically integrate closely with system services (domain controllers, single sign in systems, distributed storage, (LDAP, kerberos, AFS...)).

7.4. Inventory database

In HLT there is a lot of hardware to manage for the system administrators. Keeping an inventory of all the nodes, their contained parts and other cluster hardware, can quickly become tedious and prone to error. Luckily, collecting information from computer systems is a task that lends itself well to automation and there already exists many tools for extracting hardware information locally.

Keeping the inventory information up to date, with short delays from a change happens until it can be queried, enables the information to be used as the foundation for resources management in many forms. In the case of HLT, it is the chain configuration and task manager tools that would benefit the most from this information. One would then be able to ensure that the analysis chain is started only on nodes that are known to be in good shape. But also, if made sophisticated enough, the chain could be equipped with redundant components for fail-over purposes, thereby improving the resilience to failure in HLT.

7. Design and implementation

A simple system could be made up with a remote shell executing the appropriate command-line tools and putting the information straight into flat tables in a database. While this would most likely work and be relatively easy to implement, it might be desirable to choose an approach which organizes data in a more structured way, so that more sophisticated queries can be made. The CIM maintained by DMTF is designed exactly for the purpose of modeling the structure of distributed computer systems and is therefore a natural choice.

An inventory database is a great example of a service that would benefit from integration into a clusterapi service. Seen in conjunction with resource management, it might even be considered as a component belonging close to the core of a clusterapi solution.

This section will show how the CIM model has been implemented with the help of an ORM for storage of inventory information. Remote object manipulation will be added to the CIM+ORM combination with the intention of exploring a “remote persistent object” concept for the information transport. Finally, the service will be integrated into clusterapi so it can be managed from there and the contained information is made available for easy presentation to the user.

7.4.1. Use cases and requirements

The purpose of an inventory database is to present inventory information in a combined view compiled from all relevant hardware resources in a cluster. Further requirements are:

- Given that a hardware device has been added to the system, the information gathered must be available also when the device is unavailable for being queried for information.
- Information must be persisted over time and must survive full cluster outage, implementing persistence to permanent storage if needed.
- A solution should be designed so that functionality for adding notes - to keep a log of the history of a device - can be easily added.
- The persistence mechanism chosen should be OO-like, for it to be easier to make the CIM model in the inventory database available in a management/verification language that is close to natural languages.

The use cases for an inventory database are shown in figure 7.8.

7. Design and implementation

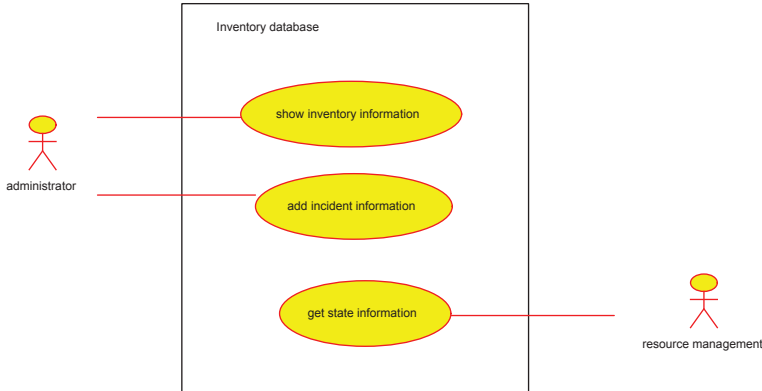


Figure 7.8.: Use cases for the inventory database.

7.4.2. Design

A centralized server for storing the information and clients submitting information to the server when it changes, is the architecture that best supports the given requirements. A peer-to-peer solution might also have been possible, but would make it harder to implement persistence.

While having the benefit of a predefined system management model in CIM, it is still necessary to evaluate it for the given application and to decide on an implementation strategy. Given the requirements, ORM was chosen as the target object persistence mechanism for its strong relational features - that should support CIM very well - and for being a mature approach, having been known and used in the software industry for a long time. Three implementation options were considered for general object persistence in HLT:

1. Use an existing CIM implementation like sblim or openpegasus and integrate it with the HLT software.
2. Create a new model to be implemented with an ORM.
3. Implement CIM with the help of an ORM.

There was initially no need for the full instrumentation that was offered in existing WBEM solutions, as only information gathering would be required for an inventory database. The investigated implementations would in this case carry functionality that would not necessarily be useful in HLT. It was also believed that being in control of the full implementation would allow for tighter integration, as well as being able to build stronger OO capabilities - and thereby a more intuitive and native feel for multiple programming languages. Finally, the prospect of being able to use a RDBS back-end

7. Design and implementation

promised an easy way for having high-availability and scalability almost for free, as these are features frequently found in database products.

The requirements of an inventory database also favors a slight architectural variation of the typical WBEM stack. These are mainly distributed, where a CIMOM runs on every node or equipment being managed by the WBEM deployment and therefore has to be “alive” for information to be available.

Whereas an inventory database requires additional persistence and query functionality, and thereby centralized features, distributed and centralized features can complement each other and lead to more flexibility for the developer, opening up for more use cases. Heavy processes requiring a lot of structure in their queries can use central resources, without affecting the rest of the system. While distributed processes can use information available locally or ask a nearby node instead of having all distributed processes communicate with a central point.

The early investigation of the use of CIM in HLT therefore concluded that existing solutions were not suitable performance- and feature-wise and that the third option would be the way to proceed.

7.4.3. Remote Persistent Objects

In coming up with an intuitive way for how the information gathering ought to work, one might envision it like this:

1. When starting up, the client asks the server for its current representation of the node in the form of an object.
2. The server returns a reference to such an object to the client. There are two possibilities:
 - a) An empty skeleton if it is the first time the client connects and there is no information on record yet for the given client.
 - b) An object populated with the information currently known by the server about the client.
3. The client then keeps the remote object up-to-date, for as long as it is running, by comparing it to the local state of the node.
4. Any operation performed locally on the object by the client should be propagated back to the server, all the way to persistent storage, due to a remote persistent object mechanism.

Since the hardware discovery client will only communicate inventory information to the hardware discovery server, this communication channel is completely private to the hardware discovery service. One can therefore choose message passing technology without having to take external requirements much into consideration.

PerspectiveBroker in the twisted framework offers, amongst other things, “translucent” access to objects. It will be as if the model in the central server is extended directly to

7. Design and implementation

the clients, where it can be easily manipulated. All while the twisted machinery takes care of keeping the traffic asynchronous. The choice of perspective broker in `hwdiscover` has partially been motivated by its highly object-oriented remote calling functionality, to see if it can support a closer to natural-language scripting/programming environment in line with the requirements.

When combined with an ORM, it becomes a powerful “remote persistent object” solution. An implementation of such a concept would propagate a change made to a remote object all the way back to persistent storage.

Remote Persistent Objects can be implemented by combining CIM, Object-Relational Mapping and Inter-Process communication. The solution presented here, use classes prepared for `sqlalchemy` to which twisted remote object decorators are added. This way, by using existing libraries, a lot of functionality has been added with little effort, while the result remains intuitive and easy to use.

7.4.4. Implementation

Two inventory database implementations have been developed in the HLT project. One python-based, following the design above, the other one written in Java, developed to be integrated into the SysMES framework. Both implementations use CIM+ORM as their internal information model and they are conceptually similar, although tailored to slightly different usage; resource management and system management.

An important motivation for choosing a solution based on CIM and ORM, was to investigate the feasibility of supporting simultaneous access to CIM information from multiple ORM compatible implementations, observing if there would be any interoperability issues between implementations written in different languages. The sophisticated ORMs available for Java and Python (`hibernate` and `sqlalchemy`) have made them good subjects for such an experiment. Having two similar implementations in different languages is also an opportunity for a comparative study of the performance and features of the two languages and the ORM packages.

The effort of designing this type of inventory gathering solution can be divided in two major tasks:

- Creating a programming language representation of a CIM model instance as well as the corresponding object-relational mapping.
- Create the mechanics/infrastructure that will automatically keep the data in the CIM model instance up to date.

The CIM+ORM mapping itself is described in detail in [225] and the design of a inventory gathering systems making use of such a mapping is first described in [226].

In CIM, there are three aspects to consider; structure, behavior and constraints. The HLT mapping is mainly concerned with structure, like data types, arrays and inheritance. Behavior (methods), except for Create, Retrieve, Update and Delete (CRUD) operations, are not strictly needed. Constraints refers to features for keeping data consistent and safe, for instance checking data types or ensuring referential integrity. These maps in many

7. Design and implementation

cases to features found in databases, but could also be implemented in a programming language if needed.

The selected ORM approach can be seen as a combination of the second and the third way of implementing exchange of management information that is described in the CIM infrastructure specification [227]. Since it is a combination of the two, it can offer shared access to the information not only by software binding to common application objects, but also at the SQL-level, if the ORM-layer is constructed in a compatible way, so that the resulting SQL from the ORM's can work on the same data set. Of course there is a third alternative in that it will always be possible to access the information by pure SQL outside the context of any ORM.

Clusterapi will be used to make inventory data accessible to clients. The top-level Cimpyp class has been equipped with a serialization method that is inherited by all CIM classes. It can serialize the data of an instantiated object to JSON for easy consumption by user interfaces.

The CIM+ORM python implementation has been given the name cimpyp. The majority of the Cimpyp source code is generated by the sbllim mof-compiler [155]. It takes MOF files as input and outputs models in a new format according to the selected back-end.

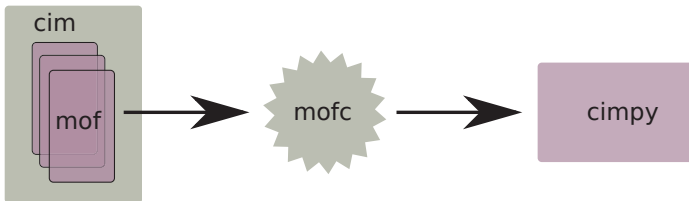


Figure 7.9.: Cimpyp generation.

A back-end has been developed that produces python source code with all the needed sqlalchemy and twisted code in place to make the concept of Remote Persistent Objects work. From the CIM MOF files, concrete classes are translated into joined table inheritance [228], while the abstract part is implemented as a normal class hierarchy without being associated with any tables. The leaf classes in the abstract class hierarchy are used as mixins to provide the columns for concrete classes according to the expected behavior in OOP inheritance.

Initially, sqlalchemy did not allow attribute overriding in a mixin inheritance hierarchy as needed for the selected approach to work. This was communicated to the sqlalchemy project along with a patch [229] that fixed the issue. The patch was accepted and included in the subsequent release [230].

Also included in the Cimpyp package is a factory module that can be used to ease object creation, and the hwinfo module that is used by the clients to look up each of the values for the CIM properties. The Unified Modeling Language (UML) diagram in figure 7.10 shows the classes included in the generation so far.

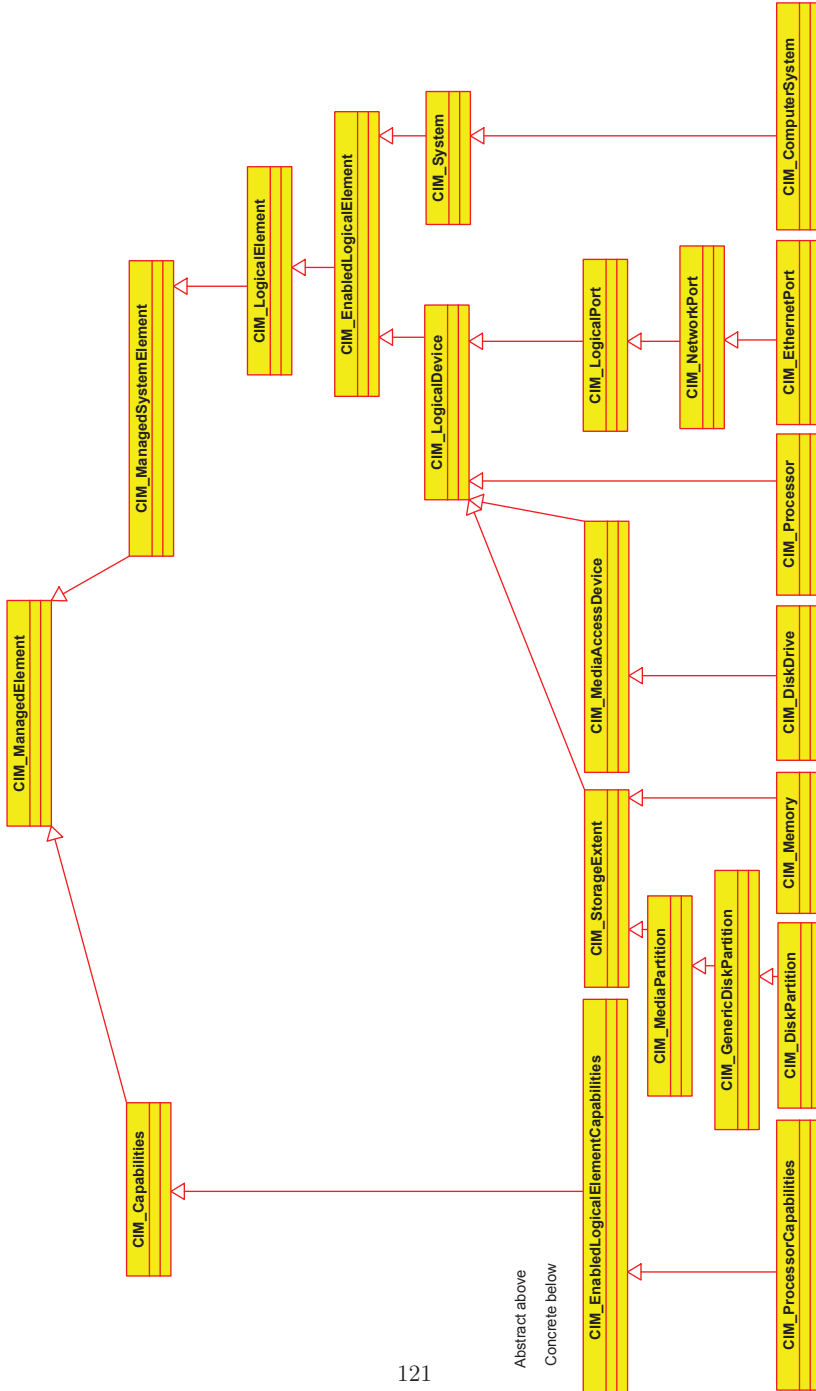


Figure 7.10.: Implemented CIM classes.

7. Design and implementation

The information gathering is implemented by two daemons, one being the central data store and the other gathering the information on the node. These are called `hwdserver` and `hwdclient`, respectively. In CIM terms, `Cimpy` is used as an object repository, but also as a storage mechanism in `hwdserver`. The client should make use of `inotify` [231] or a similar event sourced notification of system changes, to have an end-to-end push based information flow, avoiding any blocking components when propagating updates to the CIM model. The startup sequence of `hwddiscover` when integrated in `clusterapi` is shown in figure 7.11.

7. Design and implementation

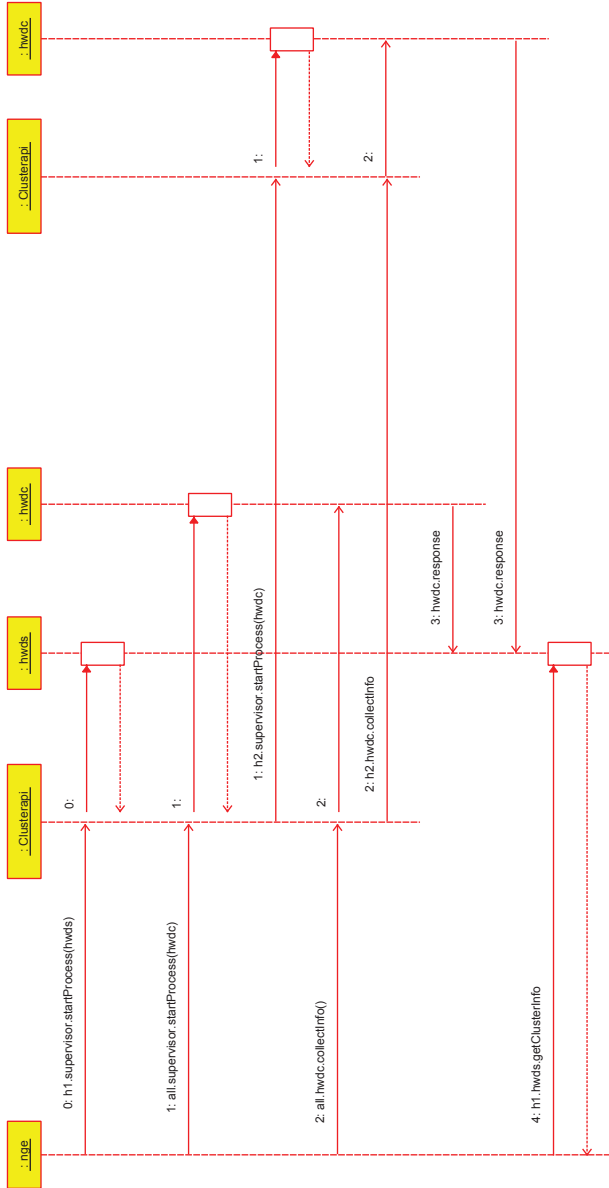


Figure 7.11.: Sequence diagram showing hwdserver and hwdclient communication when integrated in clusterapi. Actions are triggered by a third process, Nge.

7. Design and implementation

For equipment that cannot run a full client by itself, one can create fake clients that employs the appropriate protocol to bridge the communication between hwdiscover and otherwise incompatible systems. SNMP-enabled devices, for instance switches, would be a relevant example in this regard. There are also APIs defined for CIM providers. Implementing support for providers of other WBEM-stacks would allow reusing a lot of existing work.

Traditionally WBEM stacks have been implemented in a mostly distributed way, where the queries themselves are distributed from the user client to the server (CIMOM) on the device. The hwdiscover design differs in that it can be distributed and centralized at the same time. Information is gathered and stored on the nodes as it changes, while at the same time it is pushed to a central server. Information can be fetched from both kind of locations; the node from which the information originates and the central data store depending on the use case. More details on WBEM can be found in section 6.3.5.

The first stage of development of the inventory database prototype has been focused on exploring the remote persistent object concept and to prepare for java/python ORM interoperability. The second stage has been about integrating the inventory database with clusterapi. See figure 7.12 for a conceptual overview of hwdiscover and how it interacts with other systems.

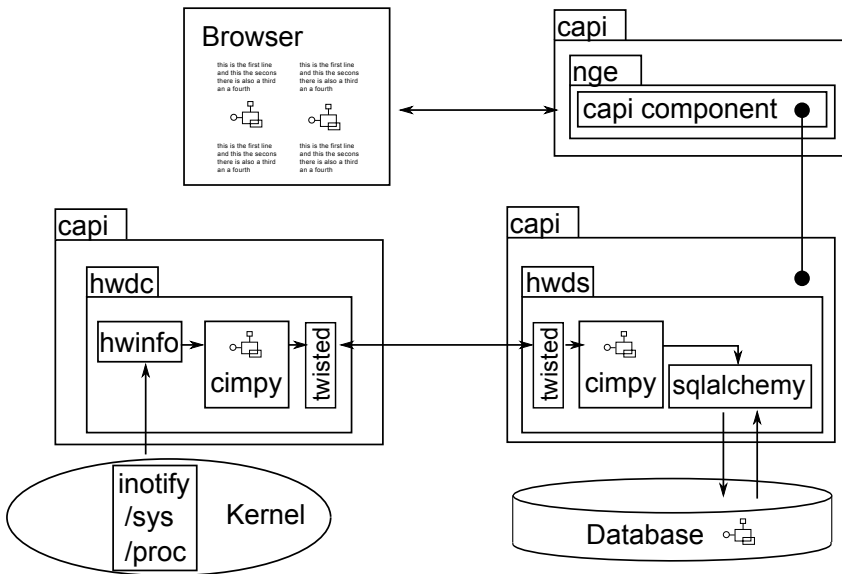


Figure 7.12.: Conceptual overview of hwdiscover and interaction with related systems.

By integrating the inventory database in clusterapi, the information-access can be made location-independent, meaning that distributed access will appear to be local to

7. Design and implementation

client code even though information might be fetched from a remote location. The actual access can either go to the central server, directly to the node in question, or to a local cache depending on which is most suitable. Centralized or direct access should have to be specified explicitly to take precedence over local cache, though.

Clusterapi should provide most of the functionality to achieve this, although a central storage will be a distinct feature of hwdiscover. In addition to being a non-volatile storage, a central object server can be used for complex queries concerned with the cluster as a whole and for querying historical data, as it will be able to keep a longer history of information than the client. Although clusterapi adds important features, the core functionality of hwdiscover should still work stand-alone for the sake of flexibility and loose coupling, in accordance with suggested guidelines.

7.5. Presentation and user interaction

It should be possible to deliver user interaction in multiple ways; graphical, web and command-line. The technology choices made so far should not exclude any of these ways of allowing the user to interact with the system.

During development and testing of clusterapi, `capictl.py`, a rudimentary CLI client has been developed that can make XML-RPC calls to the clusterapi service and print the result to the screen. It was created both to be run as a dedicated clusterapi shell with direct access to clusterapi commands, but also from the defaults shell's command-line in batch mode, so that it can be used to execute capi commands from typical bash scripts.

But most of the work related to user interaction has gone into development of a web based solution since these are easy to deploy, easy to make available and share many of the advantages of traditional GUIs. The web based solution has been given the name Nge. It is a XML/XSLT based engine that generates structured output - such as a set of web pages - for a site defined in XML. Its main feature is a built-in mechanism for interacting with python code through special tags in the XML file. During parsing, the engine will read method information from these tags, resolve the appropriate method in predefined components and finally invoke it.

The design of Nge centers around a site file, wherein an entire web application can be defined. The site file contains pages, that each corresponds to something that would logically be viewed together in a screenful. Pages can be defined inside the main site file, but can also be imported from other files, like so:

```
<page name="index" include="filepath.xml" />
```

An incoming request will typically contain a parameter defining which page should be viewed from those defined in the site file. If no page is given, the default is to render a page named `index`. On each request, the site file is parsed and processed by a SAX-parser from when hitting the start tag of the specified page until its corresponding end tag is found. The parser is implemented with a "parser controller" object that holds - at any given time - a reference to exactly one actual parser that is changing according to the context being processed. The parser events are forwarded from the parser controller to

7. Design and implementation

the currently active parser which handles the information received. One can think of it as a “linked-list context-recursive parser”, where the program flow descends parser objects recursively, while keeping a reference to the previous one, so that it can rewind as parsing completes. The involved classes can be seen in figure 7.13.

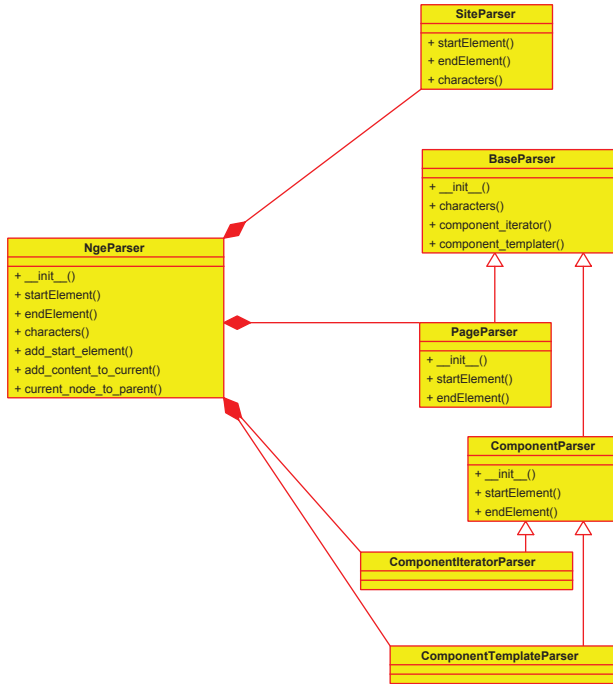


Figure 7.13.: Class diagram for nge parser.

The reason for this design is to be able to scope the parser objects to the context they are working in, so that there is less need to keep track of state in the system and synchronize state amongst parsers. The parser objects are only “alive” when they are needed and only active one at the time.

The parser controller starts by creating a SiteParser object and setting it to be the current parser. Upon being created, the SiteParser creates a new XML/DOM document that is to be built from the information read by the SAX-parser. This xml document is attached to the ParserController. A reference to the ParserController is handed over with each method call triggered by SAX events, so that through this reference, the active parser will always have access to the xml doc being built.

The site parser, upon finding the page asked for by the client, will load the page parser and set the current parser to this new object in the parser controller. The procedure is

7. Design and implementation

much the same as before, except when finding control structures or template tags, then component parsers will be activated that can load and execute components that acts as a bridge for calling python code from the site document. The page parser also handles an optional template page, that if present, will be included in all the pages defined for a site. This is handled by looking for the type attribute of a page tag and checking if it has the value “template”. The parsing ends when the end page tag for the requested page is found.

The resulting document is passed on to the last step, where it is transformed into its final rendering format (normally HTML) by a XSL style sheet. The definition of how to express the site construction is in this way made independent from the output format. One could output XML, pure HTML, HTML with CSS/javascript or pure text for that matter, given the appropriate XSLT is provided. This potentially also allows for targeting multiple devices with one site definition.

During parsing, it is the <site>, <page>, <template> and <for-each> tags that have their own context parsers. The component tags; template and for-each will be replaced with the result from the component processing and will never be seen by the XSLT processor. Other tags that don't require their own context, are just added to the document being built along the way.

The programming language definitions that are to be made available for use within a site document, are organized in python modules called, in Nge terms, components. A predefined scheme for naming and placement of files will look up a reference such as “SomeComponent.withMethod” by looking for a class called “SomeComponent” in components/somecomponent/__init__.py under the project directory. The class, if found, will be dynamically loaded and the method resolved and prepared for being executed. The methods will have to be defined according to this scheme and be present in a component, but the actual class and method definitions can be located elsewhere and imported to the component according to conventional practice in Python.

From within the XML site file, python code can be called from control structures and template helpers, like in “template” and “for-each” tags, or from buttons and links. For all these, a “call” parameter will be honored by the parser as well as optional method parameters that are matched with the signature of the method. When a call returns, python's string formatting [232] features are used to match results from components to the tags enclosed by the component tags. In addition to filling out the static tags as way of a templating system, this then also becomes the way of controlling which data is passed on from one component to another. Listings 7.3 and 7.4 shows some examples of these concepts.

Algorithm 7.3 Example of calling python code from for-each tags.

```
<!--
```

```
Example 1
```

Call `listNodes` and use the results as an arguments to `listMethods`. Assuming the result is a list, each item will be enclosed in `item` tags.

```
—>
```

```
<list>
<for-each call="ClusterApi.listNodes" return="nodename:{}">
  <item>{nodename}</item>
  <list>
    <for-each call="ClusterApi.{nodename}.system.listMethods">
      <item>{}</item>
    </for-each>
  </list>
</for-each>
</list>
```

```
<!-- The resulting XML tree could look like: —>
```

```
<list>
<item>node1</item>
<list>
  <item>someMethod1</item>
  <item>someMethod2</item>
  <item>someMethod2</item>
</list>
</list>
```

```
<!-- The XSLT translation to HTML would produce: —>
```

```
<ul>
<li>node1</li>
<ul>
  <li>someMethod1</li>
  <li>someMethod2</li>
  <li>someMethod2</li>
</ul>
</ul>
```

Algorithm 7.4 Example of calling python code from link tags.

```

<!--
Example 2
Calling startProcess for "someProcess1" on "localhost"
This would be rendered as a link labelled "start" pointing
to a page called index.
-->
<link label="start "
      call="ClusterApi.node1.supervisor.startProcess "
      arg="hwdc">index</link>

```

```

<!--
The XML tree generation and xslt translation would produce the
following HTML
-->

```

```

<a href="?page=index \
      &call=ClusterApi.node1.supervisor.startProcess \
      &arg=hwdc" target="_self">start</a>

```

Template/control structures vs. buttons/links also represents two different modes of interacting with python code. With the former, components are invoked as part of rendering of a page, while the latter will trigger the calling of the given method from an already rendered page and immediately return the user to the same page. In the latter case, if the execution of a method has changed the state and if on the same page there are components executed by templates/control structures, the content might change accordingly when the page is re-rendered. It is these two mechanisms working together that provides the mechanics for producing interactive web applications.

In the case of clusterapi, since it is based on XML-RPC, the component implementation is a simple XML-RPC proxy, only extended so that it can be pre-configured to connect to the correct port for clusterapi. Requests that it receives will be forwarded to the actual clusterapi end-point, like a transparent proxy. An overview of how interaction between nge and clusterapi is implemented is shown in figure 7.14 and the data flow in the involved systems can be seen in figure 7.15. Also relevant is the sequence diagram of hwdiscover, clusterapi and nge interaction in figure 7.11.

7. Design and implementation

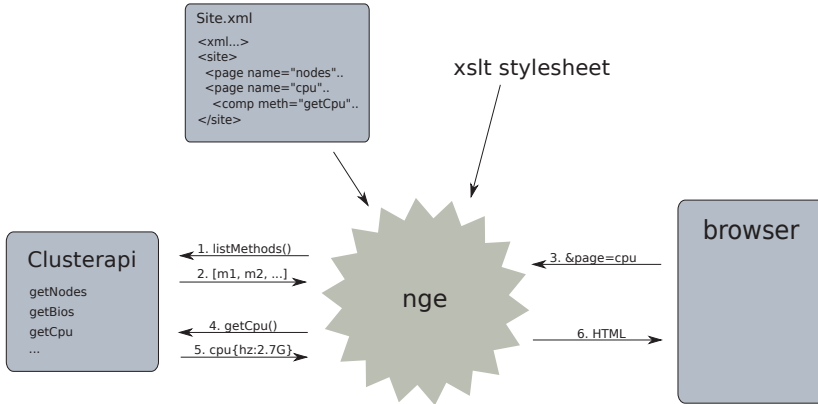


Figure 7.14.: Interaction between nge and clusterapi.

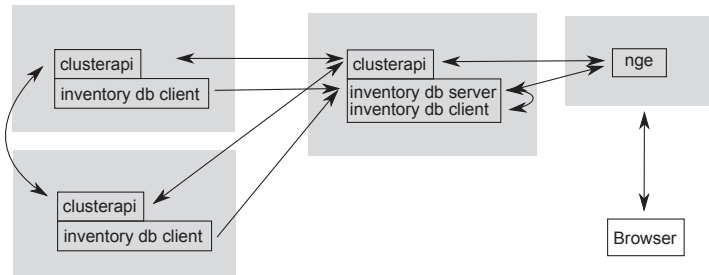


Figure 7.15.: High level overview of command and information flow in clusterapi, inventory database and nge. Communication between clusterapi instances are shown as well as information flow for the inventory database instances.

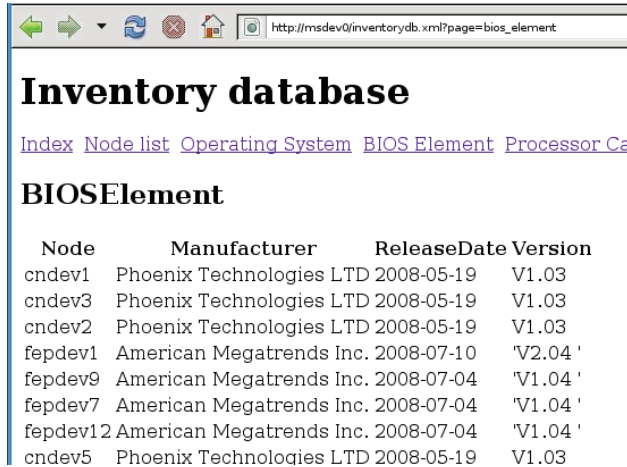
As the structure of the cimpy model is known and well-defined, it should be possible, by using service discovery and introspection, to automatically resolve and execute methods in clusterapi from nge to retrieve hwdiscover information and automatically present it in a nge site.

The work to achieve this has not been completed, but an early prototype component has been developed that uses manually defined methods made available through clusterapi to access the information in the hwdiscover database.

Algorithm 7.5 A page excerpt from a Nge site definition that will fetch BIOS information for all nodes in a hwdiscover store and present it as can be seen in figure 7.16.

```
<page name="bios_element">
  <section name="content">
    <header level="2">BIOSElement</header>
    <table>
      <row>
        <thead>Node</thead>
        <thead>Manufacturer</thead>
        <thead>ReleaseDate</thead>
        <thead>Version</thead>
      </row>
      <for-each call="ClusterApi.hwds.getBIOSElement">
        <row>
          <cell>{NodeName}</cell>
          <cell>{Manufacturer}</cell>
          <cell>{ReleaseDate}</cell>
          <cell>{Version}</cell>
        </row>
      </for-each>
    </table>
  </section>
</page>
```

In listing 7.5, an example is shown, where the hwdiscover component is used to retrieve information gathered by hwdiscover. That page definition contained in a site file would render the information shown in the screenshot in figure 7.16.



Inventory database

[Index](#) [Node list](#) [Operating System](#) [BIOS Element](#) [Processor Ca](#)

BIOSElement

Node	Manufacturer	ReleaseDate	Version
cndev1	Phoenix Technologies LTD	2008-05-19	V1.03
cndev3	Phoenix Technologies LTD	2008-05-19	V1.03
cndev2	Phoenix Technologies LTD	2008-05-19	V1.03
fepdev1	American Megatrends Inc.	2008-07-10	'V2.04 '
fepdev9	American Megatrends Inc.	2008-07-04	'V1.04 '
fepdev7	American Megatrends Inc.	2008-07-04	'V1.04 '
fepdev12	American Megatrends Inc.	2008-07-04	'V1.04 '
cndev5	Phoenix Technologies LTD	2008-05-19	V1.03

Figure 7.16.: Screen shot of the Nge inventory database.

In the same way that remote interfaces can be generated dynamically, one can also generate documentation and even mix the two in a presentation layer so as to achieve a variant of executable documentation (see section 6.5.5). For example a web page with the documentation text intertwined with buttons and other controls for actually performing tasks mentioned in the text.

7.6. Rounding out the autonomic aspects

The three preceding sub-sections describes the most complete prototypes and represents the majority of the implementation work that has been done. Their functionality have been prioritized due to being relevant from an autonomic computing and distributed management point of view, while at the same time also having relevance for the HLT cluster. Here follow the lesser developed designs, that while not having progressed as far, are nevertheless important for the full picture and a “holistic” approach.

The requirements and design of a communication layer in HLT, match up quite well with the WBEM technologies defined by DMTF, and integrating these specifications would most likely go a long way in satisfying the immediate needs in a cluster like HLT. However, one of the goals of clusterapi, is to investigate what benefits there can be from making greater use of the dynamic features found in modern interpreted languages when implementing distributed management solutions.

There is also a slight difference in focus, where WBEM technologies is more concerned by the devices produced by manufacturers and how these can easily be manipulated by providers, clusterapi is more about the domain of the application, allowing the specification of what compute and management operations should be provided in a given cluster application.

7. Design and implementation

While choosing a slightly different implementation path than WBEM, there is no reason to shy away from DMTF standards altogether as they can very well form the basis for other approaches to building distributed management systems. The CIM model for instance, has been used in the inventory solution and communication in WBEM is based on open web standards that are similar to XML-RPC used in clusterapi.

Autonomic computing and WBEM technologies are the main design drivers and the inspiration for the developed software described in this chapter. The former being the lofty vision of how large scale computing can work in the future, while WBEM/DMTF technologies are a set of very concrete standards (blueprints) for implementing management software in distributed environments. They are in this way complementary angles to working on distributed infrastructure.

In the grander scheme of autonomic computing, the immediate benefit of an inventory database and gathering system is as a source of input for intelligent process placement or resource management. An inventory database can be seen as the first step towards autonomic computing as it enables “self-consciousness” or more precisely; provides the structure for being able to reason about oneself, for a distributed computing infrastructure.

As for the core concepts of clusterapi, these align very well with the autonomic manifest in underlining the importance of unified control, as well as the importance of using open standards [233]. The resource management needs in HLT reflects the issues addressed by the topic of self-configuration [234]. Resource management with intelligent process placement can be built on top of the automated inventory gathering found in hwdiscover and clusterapi. Finally, nge relates to the presentation and control aspect of autonomic computing.

If the inventory database is the inner self-knowledge of an autonomic system, clusterapi the definition of possible actions in a cluster (including getting inventory information), then resource management is the enforcing of a policy about resource priorities in the cluster. Answering the question about who/what should have access to a given resource at any given time.

Efficient resource usage - in terms of peak performance, nominal throughput and up-time - is important for reaching the design goals of HLT with the given investment in equipment. In HLT, information from resource management systems could be used to aid the chain configuration in setting up and operating a running chain in a timely manner, with as little manual intervention as possible. High availability would be required, so the system would have to be able to quickly and automatically adjust the resource information - for instance in case of a failing node or hardware - that is fed to the live configuration systems of the physics application.

Another use case would be during development when testing analysis chains on common infrastructure. The developer must then be able to ask for the needed resources and to specify if it should be exclusive access or not. Here, exclusive access would be needed for example for performance testing when there should be no interference that could affect the benchmark results. Shared access would allow several users to access the same resources simultaneously, which should not cause problems for i.e. functionality testing during development. One could also imagine finer grained access policies, such

7. Design and implementation

as sharing the computing power, while for instance proxies to other services and port ranges are exclusive to one user.

Algorithm 7.6 Example of fictitious command-line interaction. Similar operation should also be available from a web interface. In both cases, the interaction should be done towards the same service interface.

```
$ capi resource list nodes:
    -> list available nodes, highlighting those available
$ capi resource reserve nodes 2
$ capi chain status:
    -> taskmanager and ecs-proxy is off
$ capi chain prepare/start:
    -> start proxy and runmanager with correct parameters
$ capi chain configure:
    -> configure the chain with the available resources
$ capi chain start-tools:
    -> start ecs-gui, infologger and tmgui
```

Logging is an important existing source of information that can be used as a feedback mechanism; as a way of learning about ones behavior and interaction with the surroundings. It is such feedback that can allow knowledge to evolve and heuristics to be employed in an autonomic system.

As suggested by autonomic computing, from simple, less sophisticated concepts, one can over time build greater intelligence, that ultimately can result in verification systems and languages. When having defined the application domain in a clusterapi, it will be a good foundation for building a generic verification module that can be used to ensure that the cluster is in a consistent, good state. The methods and services made available in the clusterapi could be used as the foundation for a vocabulary (the nouns) in a language to express rules.

Examples of how such verification rules could look like:

```
All <nodes> should have the same hardware configuration
    except <Node.memory> of <node> which should be 12 GB.
All volumes should have RO/RW on same machine/partition
    and RO on different machine
All <nodes> should only use RO volumes during run.
```

Figure 7.17 shows how such a module could be built on top of clusterapi.

7. Design and implementation

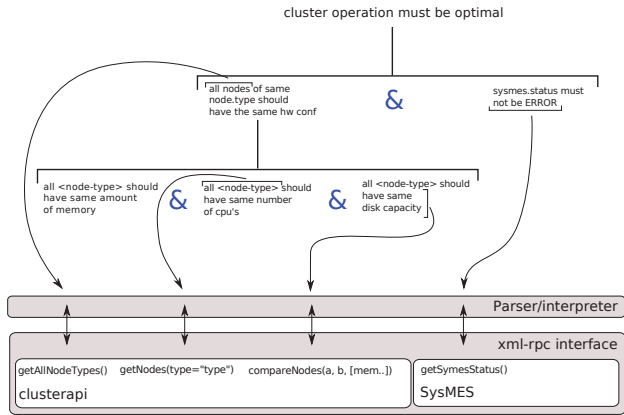


Figure 7.17.: Example of how a verification language could look.

8. Results and testing

This chapter starts with a summary of the implementation status of the prototypes, to make it clear which parts of the designs are considered implemented and which parts are still only ideas. Then follows a section on metrics and artifacts, before the main implementation delivery is attempted verified through an acceptance test. Finally there is a section on performance testing for packages where such testing has been performed.

8.1. Implementation status

The prototypes are implemented as described in the design and implementation chapters and are as such considered complete with the exceptions mentioned in this section.

8.1.1. Clusterapi

Method dispatch in clusterapi supports calling specific local or remote nodes, but calling a method on “some node” according to an algorithm as described in 7.3.4.5 is not yet in place. No big challenges is foreseen though, in implementing the algorithm described for location abstracted method dispatch. In fact, one could imagine several more types of method calling such as those shown in table 8.1.

method “url”	suggested call target
Currently implemented:	
ClusterApi.node1.listNodes	given node
ClusterApi.listNodes	local node
Envisioned for the future:	
ClusterApi.listNodes	one node according to algorithm
ClusterApi.localhost.listNodes	local node
ClusterApi.all.listNodes	all nodes
ClusterApi.[node1, node2].listNodes	all specified nodes
ClusterApi.[x].listNodes	any x number of unspecified nodes

Table 8.1.: Implemented and envisioned future method call types in Clusterapi.

8.1.2. Cimpy

The provided CIM implementation is already large enough to provide useful information for cluster management and flexible enough that accumulated information can be viewed

8. Results and testing

(i.e. total amount of memory of all nodes) as well as tabular views along different axes; information from all nodes for a given hardware device, and information of all hardware devices for a given node.

There are however more parts of the cimpy package that could be automatically generated, such as the factory helpers. Also certain aspects of the automated OR-mapping could be improved. In particular better handling of polymorphic relationships is currently the greatest hindrance to automatically translate larger parts of the CIM.

The remote persistent objects currently uses “referencable” objects in twisted parlor. Amongst the translucent object types there is also a cached variant that would be more suitable for the idea of remote persistent objects. A cached version would also presumably be more efficient.

8.1.3. Hwdiscover

The inventory database has reached the first stage of automation, where data collection is triggered when the service is started - for instance upon each boot - and activity eventually ends as the gathering is completed by the client. The method to trigger collection on the clients is exposed in clusterapi, so the next step would be to periodically trigger information gathering or enable UI control through Nge.

Ultimately, the goal is, as mentioned earlier, to trigger updating of information in the hwdiscover system by continuously listen to system change events from sources like inotify. A heartbeat mechanism would also be helpful for indicating which nodes are online and which are not. With these steps completed, the inventory database could already be used as a rudimentary resource manager, although without any authentication and authorization beyond access to the cluster.

8.1.4. Nge

From Nge one can currently call predefined methods in hwdiscover that are exposed by clusterapi, but remembering that the structure of CIM is well-known, it would be relatively easy to expose query functionality directly in Nge, so that one could do something like:

Algorithm 8.1 General query for hwdiscover ORM store.

```

<!--
    Calling the query method with a query that will return
    all computer systems.
-->
...
<for-each
    call="Clusterapi.hwds.query"
    query="CIMComputerSystem">

    <item>{NodeName}</item>

    <!-- or even better? -->
    <item>{CIMComputerSystem.NodeName}</item>

</for-each>
...

```

8.2. Developer tooling, metrics and artifacts

Unit tests, code coverage and static code analysis have been prepared for all the software packages presented in the implementation chapter. Typical tools were used for these tasks, like nose [235] for unit testing, coverage [236] for code coverage and pylint [237] for static analysis. Exceptions were made for unit testing of Hwdiscover and clusterapi, which uses trial [238], twisted's own system for unit testing, due to the asynchronous nature of twisted.

Ohcount, used here for counting source code lines, is similar to sloccount, although more recent. It supposedly has better support for newer languages. It is used by the <http://www.ohloh.net/> web page for code analysis of the projects listed at the site. It counts comments and blanks, as opposed to sloccount, and can show these separated or summed up in a total count, giving a maybe more accurate view of a project.

The results produced by these tools can be seen in section 8.2. Fabric was used for running the tools and for documenting their usage in line with the section about executable documentation (6.5.5).

All these tools have been purposely employed with automated testing and continuous integration in mind, although no fully automated system has yet been prepared. Writing unit tests along the way have helped structure and organize the code in a testable and modular way.

8. Results and testing

Metric/artifact 2012	cimpy	hwdiscover	clusterapi	nge
Unit test tool	nosetest	trial	trial	nosetest
Test coverage	76%	96%	74%	81%
Static code analysis (pylint)	-5.05/10	-5.00/10	3.34/10	0.37/10
ohcount	12321	1800	4850	8032
Metric/artifact 2014	cimpy	hwdiscover	clusterapi	nge
Unit test tool	nosetest	trial	trial	nosetest
Test coverage	76%	73%	67%	91%
API docs (add coverage?)	sphinx	sphinx	sphinx	sphinx
Static code analysis (pylint)	-3.15/10	0.30/10	3.46/10	4.56/10
ohcount	13397	3186	8712	6507

Table 8.2.: Tables showing the latest unit test, coverage and lint results for cimpy, hwdiscover, clusterapi and nge. Extracted on two occasions, the first one on 14.04.2012 and the second during june 2014.

When compared to HLT software (see for instance section 5.6), we see that the metrics and artifacts presented here are in general in a much better shape. It is still difficult to say anything about the relative code quality, though, in particular considering the considerable amount of testing the HLT software has received through commissioning and production time. Also, metrics such as these say nothing about suitability for its designed purpose or about robustness of the design.

Presumably, though, the development process has been made easier with these artifacts in place and one could expect it to be quicker for new developers to become productive in an environment with stronger support from artifacts, although the major productivity factor is likely to be the difference in project size and complexity.

8.3. Full scenario integrated acceptance test

The results presented in chapter 5 and the suggestions made in chapter 6 are core deliverables for the work in this dissertation as are the prototypes described in the implementation chapter. To verify that the prototypes created meets the requirements set forth, one has to define a reasonable acceptance criteria.

Unit tests effectively document working functionality of small pieces and with high test coverage one can be quite confident that most functionality will work as expected. For larger parts and for systems interacting with each other, functional requirements can be verified by integration tests and acceptance tests, whose goals are to determine if the functionality of the use cases are working as expected.

In the implementation chapter, the operation of the prototypes is described in detail through sequence diagrams, but always in isolation so as to keep the focus on the functionality being described in the surrounding text. A way to demonstrate that requirements have been met then, is to define a test sequence that covers a full life-cycle session of all the prototypes working together and stepping through the core use cases

8. Results and testing

shown in figures 7.1 and 7.8, in the core API listed in 7.3.5. It is this approach that is attempted below in the test script and the accompanying log.

A test script has been created to execute and document the acceptance test. It uses Vagrant [239] to create an environment with multiple virtual machines that can be used for realistic testing of multi-node interaction.

capictl.py (section 7.5), that has been developed for general testing, is used by the script to execute clusterapi methods on the command-line. To demonstrate user interaction emanating from the the UI in Nge, a head-less webkit engine [240] is used to programatically simulate browser navigation and to store screenshots of the states as the test execution progresses.

The script was implemented with fabric [241] to make it easier to group related parts and to be able to run those said selected parts in isolation during development. Fabric also conveniently labels the output with node name and whether the given line is input or output, making it easier to follow. A listing of the full test script can be found in appendix G.1.

In plain english, the sequence is as follows:

1. Base clusterapi startup on single node. Relevant figures: 7.2 and 7.3.
 - a) The supervisor is started with a configuration that will also start the clusterapi service and a hwds service.
 - b) When the supervisor and hwds service signals that they are ready, clusterapi will inspect them, and make their methods available under two sub-handlers.
 - c) The clusterapi instance is announced to the network through service discovery.
2. Start of a service with the help of the capictl.py command line tool. Relevant figures: 7.3.
 - a) The hwdc service is started by the command line client calling the startProcess method of supervisor that has been exposed by clusterapi.
 - b) When the hwdc services signals that its ready, clusterapi will inspect it and add its methods to a new sub-handler.
 - c) The hwdc services itself, collects information when started and sends it to the hwds service.
3. Start Nge and emulate a user navigating the UI and showing information. Relevant figure: 7.14.
 - a) The Nge service is started from the command line in the same way as the hwdc service. It has no methods that are exposed through clusterapi, so there is no introspection.
 - b) Navigation to the node list in the clusterapi-hwdiscover section is emulated and a screenshot made showing one node listed.
 - c) Navigation to the clusterapi-supervisor page is emulated and a screenshot made showing the current state of the services known to supervisor.

8. Results and testing

4. Start second clusterapi instance, making it a distributed configuration. Relevant figure: 7.4.
 - a) The second node starts in the same way as the first one.
 - b) The second node will then notice that there exists another clusterapi service on the network and proceed to browse it and add the remote methods to its own registry of methods.
 - c) The first node will likewise notice the presence of another clusterapi instance and will also browse the remote instance and add methods to its own registry.
5. Start hwdc service on second instance through Nge UI. Relevant figure: 7.11.
 - a) The hwdc service is started and introspected in the same way as in earlier steps.
 - b) Same navigation as in previous step is done to show that there now is two nodes in the node list and that there are now two nodes with their respective services listed in the supervisor view.

The log of running the full script is shown in listing G.2. A process tree is used to show how processes are organized under supervisor and clusterapi method calls to show how the API dynamically changes as a response to the actions performed on it. The screenshots produced by the test script is shown in chronological order in figures 8.1, 8.2 and 8.3.

8. Results and testing

[Test pages overview](#)

Clusterapi - hwdiscover

Testing hwdiscover through clusterapi.

[Index](#) [Node list](#) [Operating System](#) [BIOS Element](#) [Processor Capabilities](#) [Memory](#) [Disk](#)
[Partitions](#) [Disk Drive](#) [Ethernet ports](#) [Processors](#)

List of nodes on cluster

- node1.local

Supervisor dashboard using clusterapi

[Test page index](#)

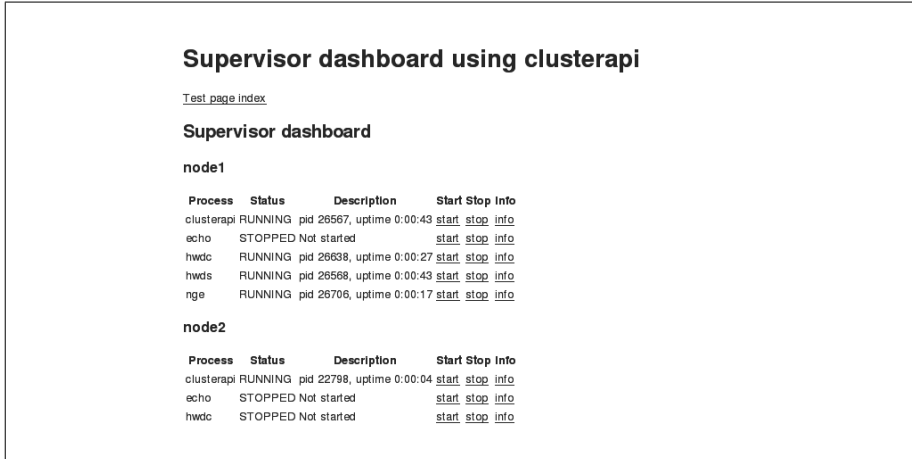
Supervisor dashboard

node1

Process	Status	Description	Start	Stop	Info
clusterapi	RUNNING	pid 26567, uptime 0:00:29	start	stop	info
echo	STOPPED	Not started	start	stop	info
hwdc	RUNNING	pid 26638, uptime 0:00:13	start	stop	info
hwds	RUNNING	pid 26568, uptime 0:00:29	start	stop	info
nge	RUNNING	pid 26706, uptime 0:00:03	start	stop	info

Figure 8.1.: Screenshots showing status information right after Nge has been started.

8. Results and testing



Supervisor dashboard using clusterapi

[Test page index](#)

Supervisor dashboard

node1

Process	Status	Description	Start	Stop	Info
clusterapi	RUNNING	pid 26567, uptime 0:00:43	start	stop	info
echo	STOPPED	Not started	start	stop	info
hwdc	RUNNING	pid 26638, uptime 0:00:27	start	stop	info
hwds	RUNNING	pid 26568, uptime 0:00:43	start	stop	info
nge	RUNNING	pid 26706, uptime 0:00:17	start	stop	info

node2

Process	Status	Description	Start	Stop	Info
clusterapi	RUNNING	pid 22798, uptime 0:00:04	start	stop	info
echo	STOPPED	Not started	start	stop	info
hwdc	STOPPED	Not started	start	stop	info

Figure 8.2.: Screenshot showing a new node appearing in the supervisor dashboard after the second clusterapi instance have been started.

8. Results and testing

The figure consists of two screenshots of a web-based supervisor dashboard. The top screenshot, titled "Supervisor dashboard using clusterapi", shows a "Supervisor dashboard" for "node1" and "node2". Each node has a table of processes with columns for Process, Status, Description, Start, Stop, and Info. In node1, clusterapi, hwdc, and rge are running, while echo is stopped. In node2, clusterapi and hwdc are running, while echo is stopped. The bottom screenshot, titled "Clusterapi - hwdiscover", shows a navigation menu and a "List of nodes on cluster" containing "node1.local" and "node2".

[Test page index](#)

Supervisor dashboard

node1

Process	Status	Description	Start	Stop	Info
clusterapi	RUNNING	pid 26567, uptime 0:00:45	start	stop	info
echo	STOPPED	Not started	start	stop	info
hwdc	RUNNING	pid 26638, uptime 0:00:29	start	stop	info
hwds	RUNNING	pid 26568, uptime 0:00:45	start	stop	info
rge	RUNNING	pid 26706, uptime 0:00:19	start	stop	info

node2

Process	Status	Description	Start	Stop	Info
clusterapi	RUNNING	pid 22798, uptime 0:00:06	start	stop	info
echo	STOPPED	Not started	start	stop	info
hwdc	RUNNING	pid 22800, uptime 0:00:02	start	stop	info

[Test pages overview](#)

Clusterapi - hwdiscover

Testing hwdiscover through clusterapi.

[Index](#) [Node list](#) [Operating System](#) [BIOS Element](#) [Processor Capabilities](#) [Memory](#) [Disk](#)
[Partitions](#) [Disk Drive](#) [Ethernet ports](#) [Processors](#)

List of nodes on cluster

- node1.local
- node2

Figure 8.3.: Screenshots showing how status has changed after the hwd client has been started on the second clusterapi instance.

The test log shows that the execution of the acceptance test executed as expected and as such it should validate the design to a satisfying degree. For even stronger verification, one could for instance do a code review. Otherwise, confidence in the reliability of a design is something that is built over time, perhaps with the help of extended tests in varying scenarios, and of course when observed in production.

8.4. Performance and scalability

For hwdiscover, testing has been done in several environments, from the local development machine to the HLT production cluster. Key numbers for the various systems are listed in table 8.3.

Early tests on the development cluster showed that it takes approximately 10 seconds to gather all information from the 20 included nodes. The init script was started in parallel on the nodes with the help of a distributed shell utility. When collection is run in parallel like this, the load on the central hwdiscover store node will saturate a single CPU core, while when running in serial, it will stay around 30%. For the individual clients, collection is completed within a second including start and stop of the service daemon. The load during this time is at most 20% of a single CPU core.

Location	Nodes	Time	CPU load	Memory	File size
dev cluster serial	8 + 1	~23 seconds	32%	0.2%	
dev cluster parallel	8 + 1	~13 seconds	98%	0.2%	
hwinfo (lower bound)	~200	7.3 seconds			1.3 MB
prod cluster	~200	~22 minutes			3.8 MB

Table 8.3.: A table showing number of nodes, time to complete, CPU load, memory usage and file size for the inventory database in different testing environments when collecting all inventory information. All measurements are performed with the top command line tool .

The last test on the production cluster was conducted in mid-May 2013 just days before the cluster was shut down. The latest version of cimpy and hwdiscover was installed on the nodes where the client were to be run and hwdiscover server was setup on a development server, a powerful node functioning as a workstation for people working on the cluster. Sqlite with files stored in /tmp was used as database backend for the hwdiscover through cimpy. Clusterapi and nge was also installed alongside the server to allow for browsing the information collected.

Ubuntu packages were prepared for all the software, including an updated version of sqlalchemy needed for cimpy to work properly on the eventually aging Ubuntu LTS release installed on the cluster.

In order to have a control measurement - a lower bound for total information gathering time - hwinfo was run stand-alone by the help of a distributed shell (dsh) on all nodes, only extracting information, printing it to screen and piping the output to a file. This took about 7.3 seconds and resulted in a combined output of about 1.3 MB. This is without any object creation or sending of objects over the wire.

Testing hwdiscover concurrently by starting the service with dsh, worked out well for the majority of the clients, until reaching a certain number of nodes, when the rest would seemingly take forever to finish and the test had to be stopped. There was little time left for optimizing the test execution to find the threshold for when a gathering would complete without issues in optimal time. It is therefore only a sequential test (for loop

8. Results and testing

on the command line) that has been completed for all nodes.

The current implementation can handle a cluster the size of HLT without any noticeable impact on the cluster application performance. When collecting information in the HLT production cluster a single server handled about 200 nodes and the full collection took 22 minutes. The size of the resulting database was 3.8 MB as compared to the output generated by `hwinfo` output of 1.3 MB.

The large difference between running a few clients (1 takes one second and 20 takes 10 and running many ~ 200 takes about 20 minutes) clearly shows that there is an issue with scaling. A thorough investigation has not yet been done to determine the limiting factor, although earlier tests and the general impression suggests that the performance of the `hwdiscover` application as a whole is currently limited by CPU-usage on the server. As such, it is also yet to be determined if the somewhat disappointing performance is due to design problems or rather just implementation blunders. It could be that `PerspectiveBroker` and `Twisted` has been a poor choice for the implementation, although this is unlikely, as `Twisted` has been used for much tougher tasks before.

8.5. Evaluation of suggestions and validity of guidelines

In terms of evaluating the suggestions made in chapter 6 and in particular the guidelines in section 6.2.7 in the context of the implementation work done in chapter 7, the first experience that comes to mind is maybe that with FOSS. The software written in here rests heavily on existing solid open sources libraries and frameworks, leveraging many man-years worth of work.

The contribution mentioned in 7.4.4 (about attribute overriding in `sqlalchemy`) shows another important advantage of open source; one is not hindered by depending on others if problems should arise. It can always be within the developer's reach to move a project forward by resolving issues, even if they are discovered in someone else's software. In this case, when a fix was found and contributed back to the project, the solution was easily included in the main source repository and featured in the next patch release. Once included it would be carried with the package and there is no need to maintain it yourself.

A similar case was with the `bigphys` driver mentioned in section 4.4.3. Here the HLT project was able to mold the linux kernel into doing a very specific thing that would fit well with the designs decided upon. But opposed to the former case, here the source code was not shared with an upstream project and it was therefore no one to share the maintenance with.

9. Conclusion and outlook

With the launch of the LHC programme, the field of high energy physics entered the large scale computing era in full force. We are still seeing this development unfold and finding its ways in becoming an even more integral part of future experiments. Being part of such an important point in scientific computing history makes HLT an interesting case to study. The experience in designing, building and operating HLT to meet the expectations of the ALICE collaboration holds valuable lessons for future projects.

In addition to having delivered the expected physics performance it is clear from the evaluation in section 5.5 that the operational performance of HLT is on par with other sub-systems in ALICE, meaning that the main purpose has been achieved for HLT. Although current tooling and infrastructure require experts available on-site for efficient debugging if the need should arise during operation. This is not unlike most other systems, but one ambitious early vision was that of a more autonomous system that would only require remote access by experts.

The description of gradual deployment in the two first sections of chapter 5 shows signs of “big up front design” and a low awareness for keeping designs in tune with with a changing reality. Potentially being the cause of painful restructuring of infrastructure. This could in turn be caused by inexperienced personnel on-site in the beginning and/or generally low continuity of on-site personnel during commissioning. There are also indications that some aspects of the design/commissioning has been underspecified in that certain parts of the system had to be patched together with scripts.

As for the general development performance, it is difficult to make strong conclusions partly because of lack of artifacts. But some data has been extracted and compared, and while the planned HPC performance review was not undertaken, much of the required data has been collected and could be reused in a future study, supplementing the work presented in here. If nothing else, the project profile, in the shape of developer activity (section 5.6.2), resembles what one would expect in an iterative project and also corresponds well with the expected activity according to the run schedule.

Software quality is related to design and artifacts. While also difficult to evaluate, lack of artifacts such as documentation is a clear sign of technical debt, as are near non-existent code coverage. Also code duplication, one of the code smells seems very high for run-critical software. Having said that, HLT has seen extensive manual testing during commissioning and has had more than satisfying operational performance.

The suggested improvements revolve around autonomic computing and well-known software engineering practices that are deemed to be a good fit for a project like HLT. The suggestions starts with deriving a few general guidelines for large scale systems based on current trends, including:

- use readily available FOSS as building blocks

9. Conclusion and outlook

- use high level languages as far as possible
- design loosely coupled for scalability

Then follows suggestions for suitable middleware technology, a presentation of relevant specifications for management software and specific suggestions for developer tooling, in particular a continuous integration toolchain is presented to which the initial set of build files was contributed.

With this developer tooling in place, certain software engineering practices are unlocked, such as Test Driven Development, while others greatly benefit; Iterative development and Collective Code Ownership, while others again are considered almost universally useful independent of available tooling; Code review.

Closing off the suggestions are some which could be deployed with relatively little effort, such as enabling centralized logging for system services and supplemental monitoring solutions that could be installed.

The second major part of this thesis is the designs and implementations described in chapter 7. Three larger packages are presented; the clusterapi, the inventory database (hwdiscover and cimpj) and nge, all built to work together in supporting aspects of an autonomic environment. The test application being an inventory database.

Of particular novelty, is perhaps two concepts embodied in the designs:

- remote persistent objects
- the idea of using ORMs derived from a common model as a language agnostic data exchange

The implementation was tested and verified in chapter 8. The acceptance script and the resulting output can be found in appendix G. Although the defined acceptance tests have been successful, much testing and refinement remains to determine if these packages would work in a production environment and indeed if the concepts would be as helpful as intended.

A tentative goal was also to exercise, during the development part of this thesis, some of the practices suggested. It is somewhat limited what can be attempted as a single person, but as mentioned in chapter G, much tooling, metrics and artifacts have been produced in line with best practices and in the preparation for continuous integration.

Maybe the most important lesson that can be learned from the experience of building the HLT is that there is potentially a large body of experience to draw from, in an industry that has been working on large scale software systems for a long time already; that software engineering is a field from which scientific computing today could learn a lot. While at the same time one will have to acknowledge that scientific computing and experimental setups have challenges that are unique and cannot necessarily be helped by traditional software engineering. Furthermore, scientific computing will have its own experiences and knowledge to contribute to the common computing field. Either way, better communication and exchange of ideas is likely to be fruitful.

This thesis points at some of the practices, tools and visions that could be useful to apply in the context of clusters like the HLT. Autonomic computing has been an

9. Conclusion and outlook

important source of inspiration for the work herein. Designs have been presented and essential parts have been implemented as prototypes for verification.

Just as new solutions have been presented as possible improvements to existing systems in HLT in this thesis, the pace of innovation has brought yet newer software that now could be seen as ready solutions to some of the problems - or parts of problems - attempted solved by the prototypes in here.

This will always be a risk when working on a project over several years. Nevertheless it is important to see concepts worked through from start to end in a stable environment to be able to make a judgement about one's ideas. Jumping from the latest new technology to the next is going to be an everlasting chase. It is then better to evaluate after a prototype has been completed what can be modified to work with new solutions or what can be replaced with new components.

One could even consider this a most important duty for a software practitioner, that is crucial for maintaining a healthy environment where software production can reach its full potential; where the building blocks are continuously refined and craft [242] is removed without hesitation.

A. SLOC count

The COCOMO data in section 5.6.1 was generated using David A. Wheeler's 'SLOC-Count' [90]. The general configuration is also described in this section, while details, including exact command line parameters, are stored in fabric scripts in the hlt-alice source repository.

A.1. Internal packages

	cost	devs	person months	schedule months	SLOC
BCL	1540555	11	137	12	20731
control	3213758	19	285	15	38259
RFLASH	53071	1	5	4	1252
PSI2	499674	5	44	8	8112
tools	574397	6	51	9	9111
monitoring	106610	2	9	5	2239
bigphys	25963	1	2	3	690
SVN-Release-Tools	18789	1	2	3	527
Framework	7920113	35	704	20	81127
MLUC	2105702	14	187	13	26898
SimpleChainConfig2	781582	7	69	10	11777
SimpleChainConfig1	944024	8	84	10	13784
EVTREPLAY	27369	1	2	3	721
TPC-OnlineDisplay	20687	1	2	3	571
interfaces	3262304	19	290	15	38740
utilities	118254	2	10	5	2441
TaskManager	898109	8	80	10	13223
util	50084	1	4	4	1193
Components	8126391	35	722	21	82884
trunk	942463	8	84	10	13765
Utility_Software	1496076	11	133	12	20231
PSI	369842	4	33	8	6313
RunManager	171040	3	15	6	3320
H-RORC	22404582	70	1990	28	192976
TRD-Readout	107067	2	10	5	2247

Table A.1.: COCOMO numbers for individual sub-projects in hlt-alice, showing number of developers, person months, scheduled months. Cost is given in US dollars.

A.2. External packages

Package	SLOC	person months	scheduled months	devs	cost
aliroot	2716157	9679.74	81.77	118.38	108966810
infologger	11051	29.91	9.09	3.29	336677
root	1730727	6030.47	68.31	88.28	67886219
Totals	4457935	15740.12	159.17	209.95	177189706

Table A.2.: SLOC numbers for external packages used in HLT.

A.3. Programming languages

SLOCCount can also provide information about which programming languages are in use in a project. For HLT this information can be seen in table A.3. The most common languages are C++, C, Java and Python.

Language	Lines	Percentage	Language	Lines	Percentage
cpp	471571	69.05	cpp	3395408	76.17
java	97196	14.23	fortran	771564	17.31
ansic	63198	9.25	ansic	232833	5.22
python	23896	3.50	sh	30108	0.68
sh	11824	1.73	perl	11075	0.25
asm	5523	0.81	python	7891	0.18
xml	3150	0.46	php	5957	0.13
jsp	2913	0.43	tcl	1733	0.04
tcl	2436	0.36	csh	651	0.01
perl	964	0.14	ruby	553	0.01
csh	257	0.04	pascal	135	0.00
Total	682928	100	java	27	0.00
			Total	4457935	100

Table A.3.: Statistics of programming languages used in HLT. Internally developed packages to the left and external packages to the right. The tables shows actual number of lines per programming language and percentage of total number of lines.

B. Git statistics

The commands and procedures that have been used to produce the data presented in this section are captured in fabric scripts that can be found either in the same repository as this thesis or in the hlt-alice repository.

The subversion repositories was checked out locally by using the “git svn” command that allows git to work with a remote subversion repository. This allowed the use of gitstats, the tool that seemed most appropriate for gathering the data needed for the repository statistics.

B. Git statistics

name	timespan	commits	authors	changes
/	4234 days, 1:54:26	33532	34	6667663
/BCL	3969 days, 0:18:01	175	2	53526
/Components	2542 days, 2:17:46	793	2	214851
/EVTREPLAY	8 days, 18:24:01	6	1	1191
/Framework	3564 days, 2:48:15	425	2	501068
/H-RORC	196 days, 3:11:58	53	1	1312392
/MLUC	3919 days, 22:12:35	237	4	66773
/PSI	2104 days, 20:36:44	109	2	47933
/PSI2	1965 days, 20:50:15	35	3	16904
/RFLASH	517 days, 3:59:23	8	1	2173
/RunManager	492 days, 2:54:59	32	1	7967
/SimpleChainConfig1	2595 days, 3:55:09	284	2	40814
/SimpleChainConfig2	1432 days, 11:53:43	207	2	25773
/TPC-OnlineDisplay	7 days, 4:33:52	3	1	4357
/TRD-Readout	39 days, 20:33:06	2	1	2638
/TaskManager	2833 days, 18:26:33	154	1	25915
/Utility_Software	2139 days, 11:25:09	225	3	35051
/control	1318 days, 5:48:29	23330	22	2370443
/interfaces	1276 days, 1:30:23	63	5	298348
/monitoring	37 days, 1:28:48	5	1	4081
/restricted	1206 days, 23:33:38	89	5	15633
/tools	1321 days, 0:04:14	204	4	37793
/trunk	2472 days, 20:44:11	281	1	40783
/util	1759 days, 6:09:19	8	1	2700
/utilities	2181 days, 3:15:31	41	2	11685

Table B.1.: Key numbers showing the timespan and total number of; commits, authors and lines changed for the individual sub-projects of the hlt-alice repository. More information about the tools used can be found here in the appendix B.

C. Use cases

Examples of modules and operations that are relevant for the HLT cluster.

C.1. Actors

- users
 - shifters
 - experts
 - developers
 - administrators
- applications
 - core applications
 - * TaskManager/processing chain
 - * tcpdump
 - support applications
 - * SysMES
 - * Infologger
 - * Logbook
 - * Resource manager/inventory database
 - * nge
 - homer-client
- RORC

C. Use cases

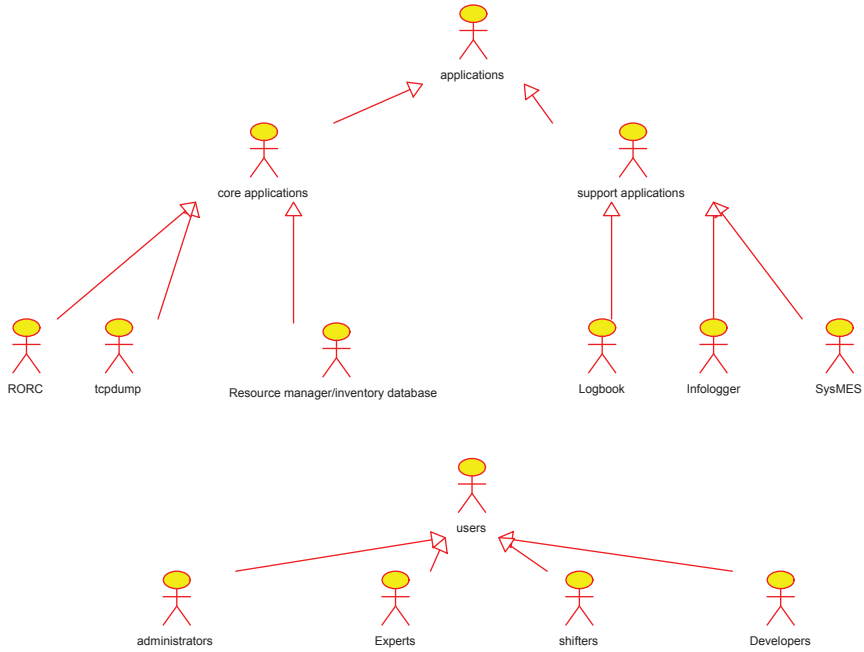


Figure C.1.: Overview of actors.

C.2. Chain operation

Run-control tools need to be able to give the TaskManager commands. ESMP needs to retrieve status information from the TaskManagers.

Operations:

```
get taskmanager status  
start/stop processing chain
```

C. Use cases

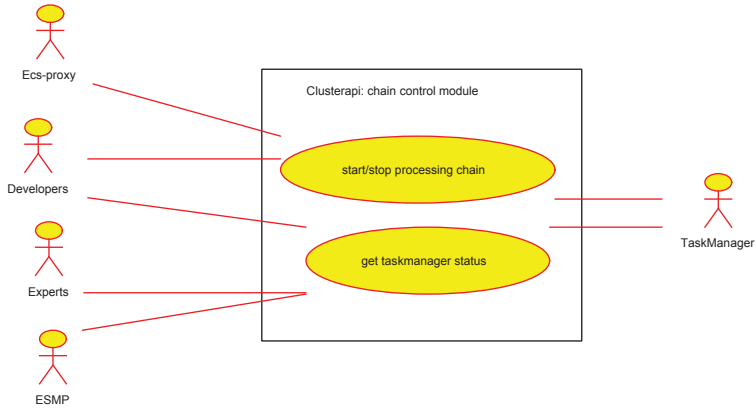


Figure C.2.: Use case diagram of chain operations.

C.3. Resource management

Resource management is something that touches the entire system. It should be used to make sure that only healthy machines are used during operation. It could be used to make idle machines in the production cluster available for other tasks. It should be used to manage the development cluster.

Behind resource management there should be an automatically updated inventory database.

- Chain configuration tools will need to know about available resources when deciding where a process should be run, so that only healthy machines are used during data taking.

Operations:

```
nodes with rorc
nodes with gpu
cpu count in node
memory amount in node
```

- For properly testing new code, the HLT has a development cluster. This cluster is used for many different types of task. From running smaller test chains to batch jobs to simple debugging and testing. Need to automate the management of such resources.

Operations:

C. Use cases

```
list vacant nodes
reserve nodes
non-exclusive reserve nodes
release nodes
```

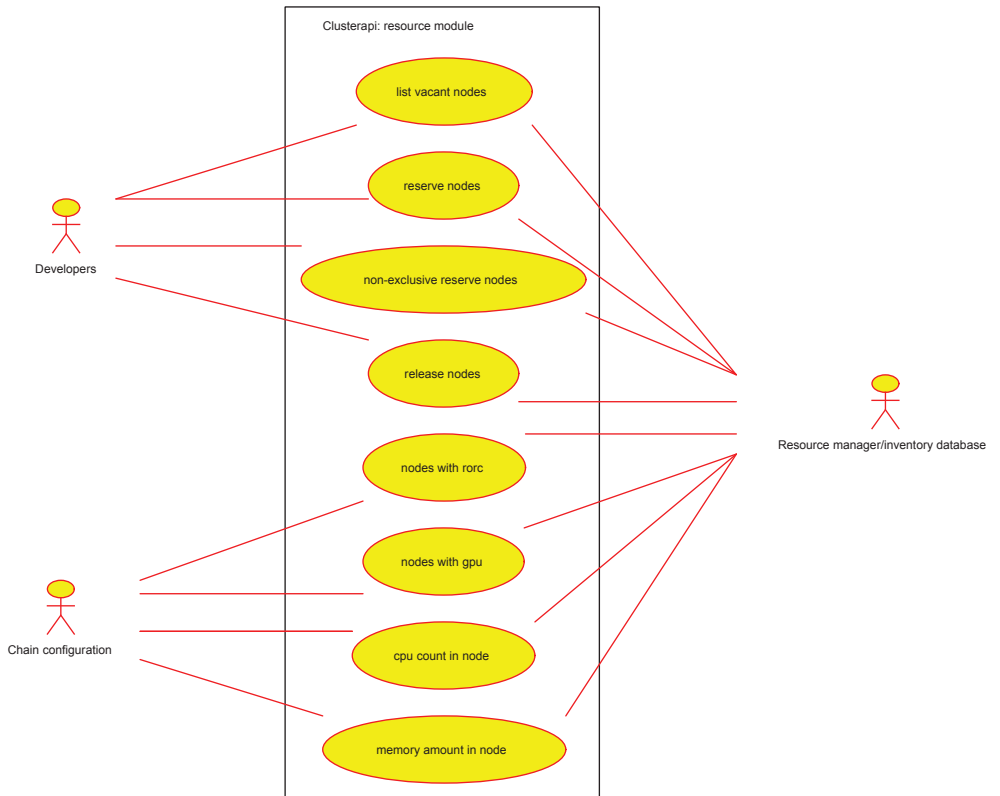


Figure C.3.: Use case diagram for resource management.

C.4. Cluster administration/Service management

These are common tasks for a system administrator to perform in any computer system. Depending on type of user and cluster, a user will want to be able to:

- Authenticate to the system.
- Be authorized for using resources.

C. Use cases

- This normally means access to the file system so that files can be stored in a home directory.
- But it can also mean the use of compute resources, such as a certain amount of nodes or CPU/memory/disk for a given amount of time.

Operations:

add/remove a user
add/remove a user to a group
change password for user

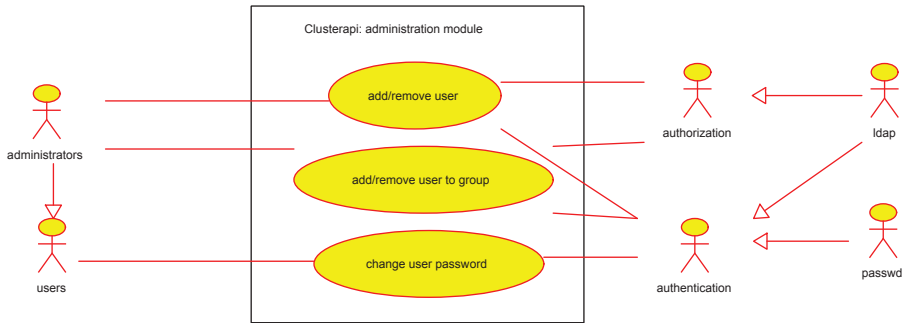


Figure C.4.: Use case diagram for cluster administration.

C.5. Data stream connection

The output of the data stream is normally a stream of serialized objects. These objects are processed and the output can for instance be histograms presented as graphics on a web page or events viewed in a 3d display.

- The TCPDumpSubscriber is the connection point for the event display and consumers of histograms that are produced by the HLT processing chain. Sending the same data to many clients is expensive. Multicast would reduce the network load.

Operations:

pub-sub connection
multicast connection

- Histograms produced by the HLT processing chain must be presented to shifters in the control room for data quality monitoring. Currently histogram data is dumped to a machine inside the cluster and a web server runs a script that creates the final histograms that is presented to the user in a browser.

C. Use cases

Operations:

view all histograms

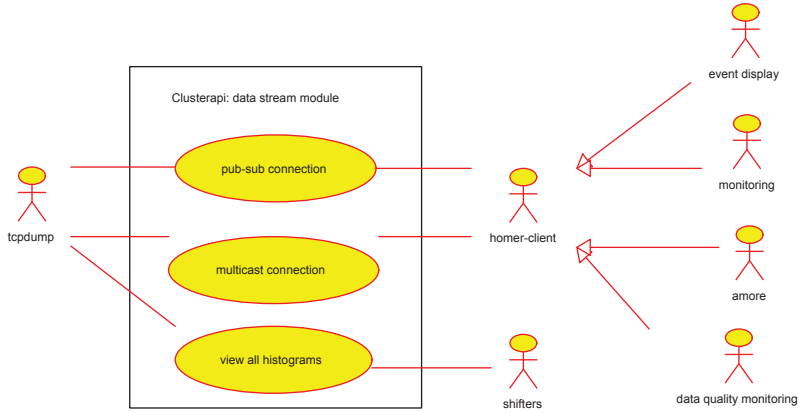


Figure C.5.: Use case diagram for data stream connections.

C.6. Statistics retrieval

Statistics for the operational performance of HLT is useful for performance evaluation. Currently this is gathered from various databases with stand-alone programs manually.

Operations:

amount of logmessages
distribution of logmessages
operational performance
meantime to failure
failure cause

The response here could either be raw data in the form of csv or even graphs and ready-made histograms.

C. Use cases

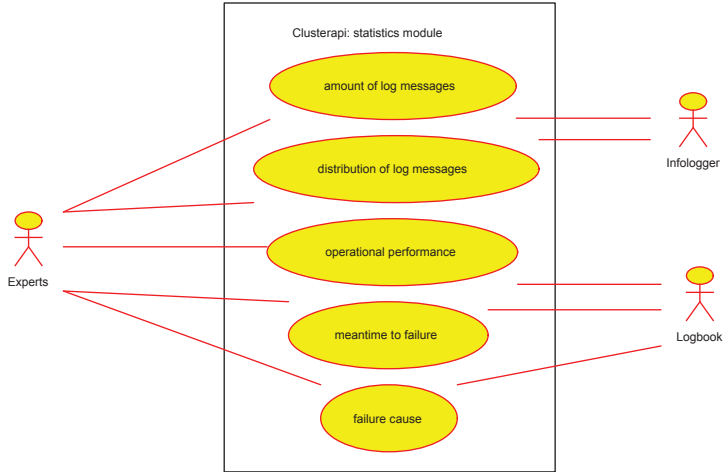


Figure C.6.: Use case diagram for statistics retrieval.

C.7. RORC control

The RORC cards can be used for playback of events from disk for more realistic testing. During normal operation, it is also important for experts to be able to query the status information.

Operations:

```
start/stop rorc replay  
load rorc replay files  
get rorc event/buffer status
```

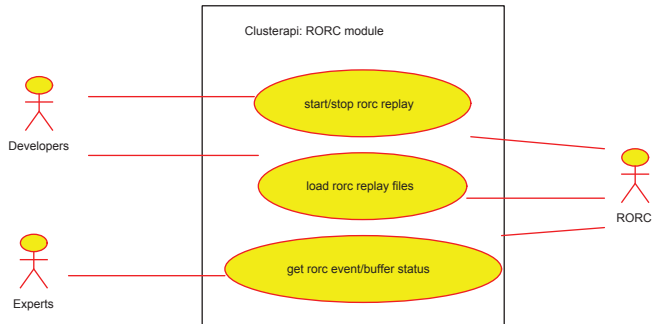


Figure C.7.: Use case diagram for RORC control.

C.8. SysMES

Run-control needs to know which state the cluster is in according to SysMES, to allow operation or not for a given user.

Operations:

```
get sysmes status
```

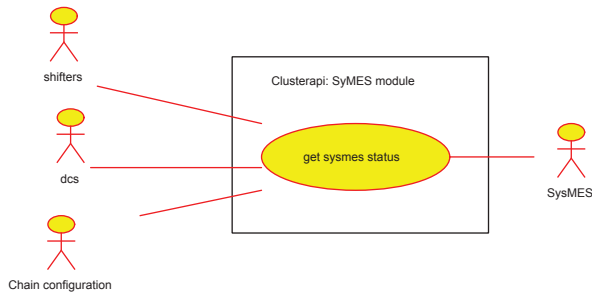


Figure C.8.: Use case diagram for SyMES operations.

C.9. Clusterbus use cases

Use cases were also gathered as input from individuals in the HLT group. These were to a great extent formulated more as wishes rather than properly justified use cases. They are nevertheless mentioned here for the sake of completeness and for showing the different perspectives on what is needed in HLT. These are for the most part related to a high performance data centric framework for routing data, as briefly mentioned in section 7.3.2.

- Should enable push. An event mechanism for the data stream in addition to poll.
- Should be possible to proxy the raw data stream.
- Should be able to handle millions of messages per seconds.
- Should be able to use multi-cast in cases where it makes sense.
- Should be possible for a client to request a subset of data it is actually interested in.
- Should be possible to secure connection points with authentication/authorization.

D. Service discovery of data sinks

Being able to tap into the transport framework of HLT and read events directly from the data flow, is useful for debugging, quality assurance or just for generating a certain type of test data. The `TCPDumpSubscriber` can be connected to any other component in the framework and dumps the data it receives to a normal TCP port from where it can be read by other applications outside the main transport framework. This is for instance used to connect the event display to the data stream so that events can be read and shown in the control room.

Since the HLT chain was foreseen to be configured dynamically, a mechanism was needed for retrieving the connection information for such `TCPDumpSubscribers`. Service discovery, as mentioned in 6.3.4, can be used to for such a tasks. The operational implementation in HLT can be summarized as:

1. The data sinks (`TCPDumpSubscriber`) start up and announce their host name, port and service name via the Avahi service discovery mechanism.
2. When the data flow in HLT starts, additional information is added to the `TXT` field of a service record about what kind of data is available on this particular sink and this information is propagated to all Avahi-enabled nodes.
3. This means that portal and gateway machines have this information available in their local Avahi daemon and can serve it over a XML-RPC interface to machines on other networks that wants to connect to the internal data sinks via port forwarding on portals and gateways machines.

There was a slight mismatch between initial and final requirements that led to sub-optimal use of service discovery technology, announcing much more information over Avahi than what it was designed for. Not only information for connecting to the service, but also information about the data provided by the given `TCPDumpSubscriber`.

In addition, Avahi is not tuned for large clusters, but rather for small home networks and has been specifically designed with strict security considerations [243] to minimize the risk of potential attacks. This means that in large networks with many nodes, Avahi won't necessarily see all of them. For this reason, the version used in HLT had to be patched in order to increase the cache size of the Avahi daemon to make it work in our setup. The patched Avahi packages were produced and distributed through local package repository with the tools provided with the distribution.

E. Log data mining

The infologger database as described in 4.4.4.2 has collected log messages throughout the runs since the very beginning. From July 2010, the regular dumps of the database were stored for later analysis instead of just being deleted. These dumps have been merged into a single database that can be used to mine for data regarding logging and error conditions in HLT. Vital information about this database is shown in table E.1.

Data set	Time span	Total number of messages	Database size in GB
1	2010-07-22 - 2010-10-29	178247483	27GB
2	2010-10-29 - 2011-12-07	202685049	32GB
Combined	2010-07-22 - 2011-12-07	380932532	59GB

Table E.1.: Overview of the data sets. MySQL[244] is used with the innodb engine and the database size is measured by the size of the message.MYD file .

The graphs of the dumps show the distribution of the various types of log messages over time (E.1, E.2).

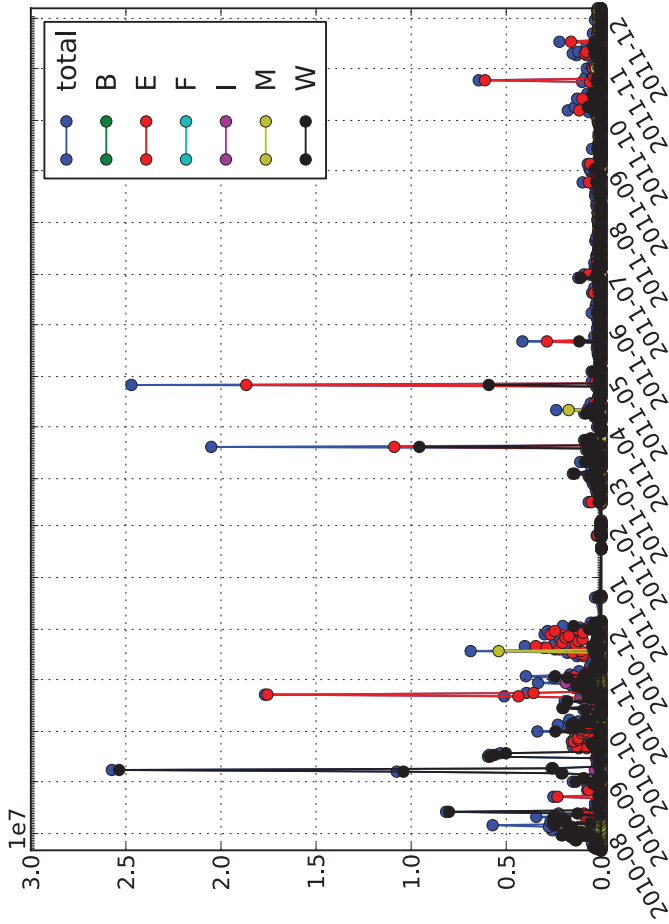


Figure E.1.: Total number of log messages per day .

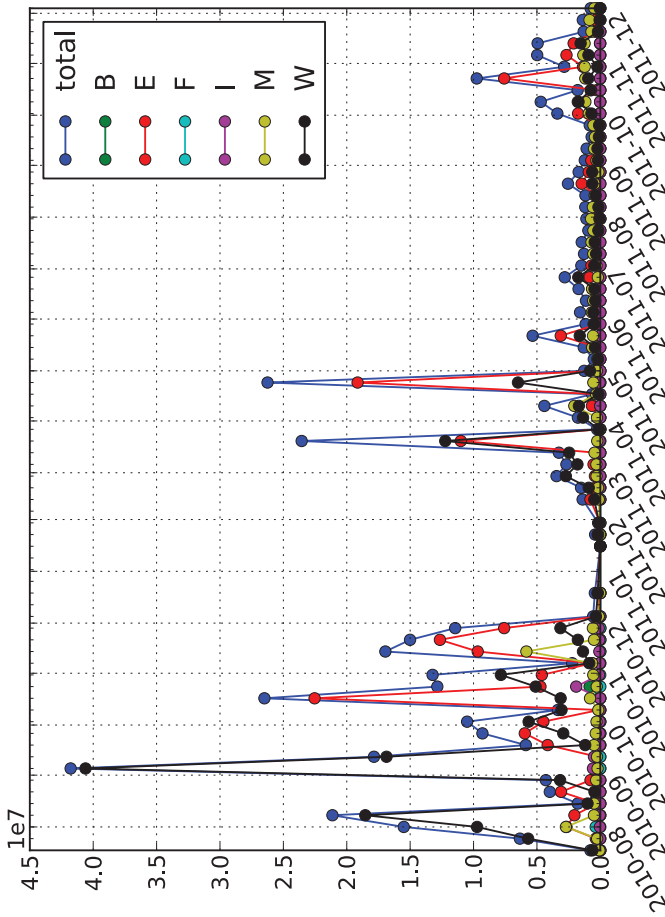


Figure E.2.: Total number of log messages per week .

Python[245], SQLAlchemy[138] and matplotlib[246] packages were used to extract data from the infologger database and create the graphs that is shown in 5.5.2. The source code for these scripts can be seen in listing E.1.

Algorithm E.1 Script that queries the infologger database for basic information.

```

1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """
5  _Created_on_Wed_Jun_1_14:05:27_2011
6  _@author:_Oystein_Senneset_Haaland
7  """
8
9  from sqlalchemy import create_engine, MetaData
10 from sqlalchemy import Table, Column, TIMESTAMP, func
11 from sqlalchemy.sql import select
12
13 connectionUrl = "mysql://infologger:gee8Chao@kallen/infologger"
14 engine = create_engine(connectionUrl)
15
16 meta = MetaData()
17 meta.bind = engine
18
19 messages = Table('messages', meta, Column('timestamp', TIMESTAMP
20         ), autoload=True)
21 hltproxy = Table('hltproxy', meta, Column('timestamp', TIMESTAMP
22         ), autoload=True)
23 meta.create_all()
24
25 def numberOfElements():
26     result = messages.count().execute()
27     return result.scalar()
28
29 def timeSpan():
30     from datetime import datetime
31     query = select([
32         func.MIN(messages.c.timestamp).label("startDate"),
33         func.MAX(messages.c.timestamp).label("endDate")
34     ])
35     row = engine.execute(query).first()
36     return datetime.fromtimestamp(row.startDate), datetime.
37         fromtimestamp(row.endDate)
38
39 print "INFOLOGGER_STATISTICS"
40
41 print "\nTables_in_database:"
42 for table in meta.sorted_tables:
43     print table, ":", messages.columns
44
45 print "\nTime_span:"
46 startDate, endDate = timeSpan()
47 print "%s - %s" % (startDate, endDate)
48
49 print "\nNumber_of_elements_in_this_dataset:"
50 print numberOfElements()

```

F. Software configuration/build system

F.1. The Software packages of HLT

Package	Beginning of HLT ~2006?	Current
ROOT	Bash/make	Bash/make and CMake
AliROOT	make	CMake
EVTREPLAY	make	make
HRORC-tools		
SysMES	Ant?	
infologger	make	make and CMake
ecs-proxy	make	make
taxi	make	make
pendolino	make	make
BCL	make	CMake effort started
Components	make	CMake effort started
Framework	make	CMake effort started
MLUC	make	CMake effort started
PSI2	make	CMake effort started
RunManager	make	CMake effort started
SimpleChainConfig	make	CMake effort started
TaskManager	make	CMake effort started
Analysis framework	Had autotools? And shell scripts?	Now in AliROOT. Uses CMake

Table F.1.: Build systems in HLT software. (see docs/SoftwareCatalog/software_catalog.lyx). Where it says make, pure make is meant, not with any autotools assistance. In ROOT Bash is used configuration and make for building. Lately CMake support have been added. Infologger was originally only pure make files. With the adoption to HLT, CMake files was added .

F.2. CMake example

- CMake example from the infologger also showing how CPack and CTest functionality is enabled.
- The test scripts used for integration testing are included to demonstrate how acceptance testing can be done with CMake.

F.3. Full cycle testing of infologger

Full cycle testing (acceptance/integration testing) simulates real-life running of the system, exercising all the involved components in the software package. After modifications have been made and the source is compiled, tests were run with CTest for each test set like this:

1. Setup of test database with default infologger schema in dedicated test directory.
2. Start server and connect to database. Start client daemon and connect to server.
3. Log test messages by using included command-line tool. The messages would be delivered via the infologger library to the client daemon from which they would be sent through to the server and finally the database.
 - a) log a message or messages according to a pattern
 - b) check that pattern appears in the database
4. tear-down of test environment

G. Acceptance test

G.1. Test script

```
1
2 """
3
4 _Acceptance_test_for_clusterapi
5
6 """
7
8 import time
9
10 from ghost import Ghost
11 from contextlib import contextmanager
12
13 from fabric.api import local, env, run, task
14 from fabric.api import hosts, execute, cd, prefix
15
16
17 env.use_ssh_config = True
18 env.ssh_config_path="node_ssh.config"
19
20 env.directory = '~/git/clusterapi'
21 env.activate = 'source~/home/vagrant/.virtualenvs/capi/bin/
    activate'
22
23 ghost = Ghost()
24 base_url = "http://localhost:3030/"
25
26
27 ###
28 # Helpers
29 ###
30
31 @contextmanager
32 def virtualenv():
33     with cd(env.directory):
```

G. Acceptance test

```
34         with prefix(env.activate):
35             yield
36
37
38 def ptree():
39     """
40     Print_clusterapi_process_tree
41     """
42     print "\nClusterapi_process_tree:"
43     run("ptree_la 'supervisorctl_pid'")
44
45
46 ###
47 # Test steps
48 ###
49
50 @task
51 @hosts(["node1"])
52 def step1_start_first_capi():
53     """
54     Step_1:_Start_clusterapi_on_the_first_node.
55     """
56     with virtualenv():
57
58         print "\nStart_supervisord_with_the_clusterapi_
59             configuration."
60         run("supervisord_c_supervisord_node1.conf")
61         time.sleep(5) # Sleep until it must be
62             ready
63
64         print "After_starting_clusterapi,_show_the_current_
65             status."
66         ptree()
67
68         print "Check_that_the_core_services_are_running:"
69         assert "RUNNING" in run("supervisorctl_status_clusterapi
70             ")
71         assert "RUNNING" in run("supervisorctl_status_hwds")
72
73         print "Check_that_the_clusterapi_service_is_listed_in_
74             avahi:"
75         assert "clusterapi@" in run("avahi-browse_c_clusterapi
76             .tcp")
```

G. Acceptance test

```
72     print "List_all_the_sub-handlers_(supervisor ,_hwds)"
73     run('python_capictl.py_listSubHandlers')
74
75     print "List_top-level_clusterapi_methods_and_module_
76           introspection_methods"
77     run('python_capictl.py_system.listMethods')
78
79     print "List_sub-handler_methods"
80     run('python_capictl.py_listSubHandlerMethods_supervisor'
81         )
82     run('python_capictl.py_listSubHandlerMethods_hwds')
83
84     print "Show_that_the_hwd_database_is_empty_before_
85           calling_capi_method"
86     res_get_nodes = run('python_capictl.py_hwds.getNodes')
87
88 @task
89 @hosts(["node1"])
90 def step2_start_hwd_client():
91     """
92     Step_2:_Start_hwdc_and_show_status.
93     """
94     with virtualenv():
95
96         print "Use_capictl_to_start_hwdc_on_node1"
97         run('python_capictl.py_supervisor.startProcess_hwdc')
98         time.sleep(5) # Sleep until it must be
99                       ready
100
101         pstree()
102
103         # Check that hwdc is up and running
104         assert "RUNNING" in run("supervisorctl_status_hwdc")
105
106         # List modules, now assure hwdc is there
107         assert "hwdc" in run('python_capictl.py_listSubHandlers'
108                               )
109
110         # List methods for hwdc
111         run('python_capictl.py_listSubHandlerMethods_hwdc')
112
113         # Show that the hwd database has been updated
114         res = run('python_capictl.py_hwds.getNodes')
115         #assert len(res) > len(res_get_nodes)
```

G. Acceptance test

```
111 |
112 | @task
113 | @hosts(["node1"])
114 | def step3_start_nge():
115 |     """
116 |     Step 3: Start nge and capture screenshots to show current
117 |     status
118 |     """
119 |     with virtualenv():
120 |         run('python capictl.py supervisor.startProcess_nge')
121 |
122 |     # Show the current state (only one node)
123 |
124 |     # In the the clusterapi-hwdiscover page after navigating to
125 |     the node list page
126 |     page, resources = ghost.open(base_url + "clusterapi-
127 |     hwdiscover.xml")
128 |     page, resources = ghost.click('a[href="?page=node_list"]',
129 |     expect_loading=True)
130 |     ghost.capture_to("01_node_list_1.png")
131 |
132 |     # In the clusterapi-supervisor page:
133 |     page, resources = ghost.open(base_url + "clusterapi-
134 |     supervisor.xml")
135 |     ghost.capture_to("02_capi_supervisor_1.png")
136 |
137 | @task
138 | @hosts(["node2"])
139 | def step4_start_second_capi():
140 |     """
141 |     Step 4: Start a second clusterapi instance in a second node
142 |     """
143 |     with virtualenv():
144 |
145 |         print "\nStart supervisor_d with the clusterapi_
146 |         configuration."
147 |         run("supervisord -c supervisord_node2.conf")
148 |         time.sleep(5) # Sleep until it must be
149 |         ready
150 |
151 |     # Show the current state in the clusterapi-supervisor page
152 |     page, resources = ghost.open(base_url + "clusterapi-
153 |     supervisor.xml")
154 |     ghost.capture_to("03_capi_supervisor_2.png")
```

G. Acceptance test

```
147
148   ## Start the hwdc client on the second node from nge
149
150   # NOTE: This does not work:
151   #page, resources = ghost.click('table#node2 a#hwdc_start',
152     expect_loading=True)
153
154   # NOTE: Have to do this:
155   selector_string = "table#node2 > tr:nth-child(4) > td:nth-
156     child(4) > a"
157
158   ## Printing node information...
159   #javascript = ""
160   #{function () {
161       #var element = document.querySelector('%s');
162       #return element.nodeName;
163       #})();
164   #"#"
165
166   # Clicking something...
167   javascript = ""
168   (function() {
169     var element = document.querySelector('%s');
170     var evt = document.createEvent("MouseEvent");
171     evt.initMouseEvent("click", true, true, window, 1,
172       1, 1, 1, 1,
173     false, false, false, false, 0, element);
174     return element.dispatchEvent(evt);
175   })();
176   ""
177
178   result, resources = ghost.evaluate(javascript %
179     selector_string, expect_loading=True)
180
181   # 1. Show the difference in clusterapi-supervisor
182   ghost.capture_to("04_capi_supervisor_3.png")
183
184   # Wait for hardware discovery to complete
185   time.sleep(5)
186
187   # 2. Show the differences in the clusterapi-hwdiscover
188   page, resources = ghost.open(base_url + "clusterapi-
189     hwdiscover.xml")
```

G. Acceptance test

```
185     page, resources = ghost.click('a[href="?page=node_list"]',
186     expect_loading=True)
187     ghost.capture_to("05_node_list_2.png")
188
189     def delete_hwdiscover_db():
190         with virtualenv():
191             run("rm-f ../hwdiscover/hwdiscover.sqlite")
192
193     @task
194     def stop_clusterapi():
195         """
196     ~~~~Stop_clusterapi
197     ~~~~"""
198         with virtualenv():
199             run('supervisorctl_shutdown')
200
201     @task
202     def full_acceptance_test():
203         """
204     ~~~~Full_acceptance_test.
205
206     ~~~~Showing_first_one_node,_then_two_nodes_being_manipulated_
207         ~~~~with_clusterapi
208     ~~~~commands_from_command-line,_then_later_also_via_Nge.
209     ~~~~"""
210
211         # Start the two virtual machines.
212         #local('vagrant up node1')
213         #local('vagrant up node2')
214
215         # Preparations/cleanup of previous runs
216         execute(delete_hwdiscover_db, hosts=["node1"])
217
218         # Step 1: start the first vagrant instance (node1) and
219         # supervisor with
220         # clusterapi. Hwds is also started.
221         #local("vagrant up node1")
222         execute(step1_start_first_capi, hosts=["node1"])
223
224         # Step 2: Start the hwd client
225         execute(step2_start_hwd_client, hosts=["node1"])
226
227         # Step 3: Start nge
```

G. Acceptance test

```
226 # Here we would like to demonstrate GUI functionality with
      Nge,
227 # forinstance with the help of ghost.py to click around in
      the UI
228 execute(step3_start_nge, hosts=["node1"])
229
230 # Step 4: Start second capi instance
231 #local("vagrant up node2")
232 execute(step4_start_second_capi, hosts=["node2"])
233
234 # NOTE: Could show wind-down; stop sub-services and show the
      difference
235 # again.
236
237 # Finally full stop:
238
239 # Stop all clusterapi and virtual machine instances
240 #execute(stop_clusterapi, hosts=env.hosts)
241 #execute(stop_clusterapi, hosts=["node1"])
242
243 #local('vagrant halt node2')
244 #local('vagrant halt node1')
```

G.2. Test log

```
1 [node1] Executing task 'delete_hwdiscover_db'
2 [node1] run: rm -f ../hwdiscover/hwdiscover.sqlite
3 [node1] Executing task 'step1_start_first_capi'
4
5 Start supervisord with the clusterapi configuration.
6 [node1] run: supervisord -c supervisord_node1.conf
7 After starting clusterapi, show the current status.
8
9 Clusterapi process tree:
10 [node1] run: pstree -la 'supervisorctl pid'
11 [node1] out: supervisord /home/vagrant/.virtualenvs/
      capi/bin/supervisord -c supervisord_node1.conf
12 [node1] out: python clusterapi/clusterapi.py
13 [node1] out: twisted /home/vagrant/.virtualenvs/capi
      /bin/twisted --nodaemon --pidfile=hws_node1.pid
      hws_server --database=sqlite:///hwdiscover.sqlite
14 [node1] out:
```

G. Acceptance test

```
15 |
16 | Check that the core services are running:
17 | [node1] run: supervisorctl status clusterapi
18 | [node1] out: clusterapi                                RUNNING
19 |               pid 26567, uptime 0:00:07
20 | [node1] out:
21 | [node1] run: supervisorctl status hwds
22 | [node1] out: hwds                                    RUNNING
23 |               pid 26568, uptime 0:00:08
24 | [node1] out:
25 | Check that the clusterapi service is listed in avahi:
26 | [node1] run: avahi-browse -c _clusterapi._tcp
27 | [node1] out: +   eth1 IPv6 clusterapi@node1.local
28 |               _clusterapi._tcp local
29 | [node1] out: +   eth1 IPv4 clusterapi@node1.local
30 |               _clusterapi._tcp local
31 | [node1] out:
32 | List all the sub-handlers (supervisor, hwds)
33 | [node1] run: python capictl.py listSubHandlers
34 | [node1] out: ['node1', 'supervisor', 'system']
35 | [node1] out:
36 | List top-level clusterapi methods and module
37 | introspection methods
38 | [node1] run: python capictl.py system.listMethods
39 | [node1] out: ['listSubHandlerMethods',
40 | [node1] out: 'listSubHandlers',
41 | [node1] out: 'listNodes',
42 | [node1] out: 'listAllSubHandlerMethods',
43 | [node1] out: 'node1.methodHelp',
44 | [node1] out: 'node1.listMethods',
45 | [node1] out: 'node1.methodSignature',
46 | [node1] out: 'system.methodHelp',
47 | [node1] out: 'system.listMethods',
48 | [node1] out: 'system.methodSignature',
49 | [node1] out: 'supervisor.system.methodHelp',
50 | [node1] out: 'supervisor.system.listMethods',
51 | [node1] out: 'supervisor.system.methodSignature',
52 | [node1] out: 'node1.supervisor.system.methodHelp',
53 | [node1] out: 'node1.supervisor.system.listMethods',
```


G. Acceptance test

```
53 [node1] out: 'node1.supervisor.system.  
methodSignature']  
54 [node1] out:  
55  
56 List sub-handler methods  
57 [node1] run: python capictl.py listSubHandlerMethods  
supervisor  
58 [node1] out: ['clusterapi.getProcessXmlrpcUrl',  
59 [node1] out: 'supervisor.addProcessGroup',  
60 [node1] out: 'supervisor.clearAllProcessLogs',  
61 [node1] out: 'supervisor.clearLog',  
62 [node1] out: 'supervisor.clearProcessLog',  
63 [node1] out: 'supervisor.clearProcessLogs',  
64 [node1] out: 'supervisor.getAPIVersion',  
65 [node1] out: 'supervisor.getAllConfigInfo',  
66 [node1] out: 'supervisor.getAllProcessInfo',  
67 [node1] out: 'supervisor.getIdentification',  
68 [node1] out: 'supervisor.getPID',  
69 [node1] out: 'supervisor.getProcessInfo',  
70 [node1] out: 'supervisor.getState',  
71 [node1] out: 'supervisor.getSupervisorVersion',  
72 [node1] out: 'supervisor.getVersion',  
73 [node1] out: 'supervisor.readLog',  
74 [node1] out: 'supervisor.readMainLog',  
75 [node1] out: 'supervisor.readProcessLog',  
76 [node1] out: 'supervisor.readProcessStderrLog',  
77 [node1] out: 'supervisor.readProcessStdoutLog',  
78 [node1] out: 'supervisor.reloadConfig',  
79 [node1] out: 'supervisor.removeProcessGroup',  
80 [node1] out: 'supervisor.restart',  
81 [node1] out: 'supervisor.sendProcessStdin',  
82 [node1] out: 'supervisor.sendRemoteCommEvent',  
83 [node1] out: 'supervisor.shutdown',  
84 [node1] out: 'supervisor.startAllProcesses',  
85 [node1] out: 'supervisor.startProcess',  
86 [node1] out: 'supervisor.startProcessGroup',  
87 [node1] out: 'supervisor.stopAllProcesses',  
88 [node1] out: 'supervisor.stopProcess',  
89 [node1] out: 'supervisor.stopProcessGroup',  
90 [node1] out: 'supervisor.tailProcessLog',  
91 [node1] out: 'supervisor.tailProcessStderrLog',  
92 [node1] out: 'supervisor.tailProcessStdoutLog',  
93 [node1] out: 'system.listMethods',  
94 [node1] out: 'system.methodHelp',
```

G. Acceptance test

```
95 [node1] out: 'system.methodSignature',
96 [node1] out: 'system.multicall']
97 [node1] out:
98
99 [node1] run: python capictl.py listSubHandlerMethods
    hws
100 [node1] out: <twisted.python.failure.Failure <class '
    xmlrpclib.Fault'>>
101 [node1] out:
102
103 Show that the hwd database is empty before calling
    capi method
104 [node1] run: python capictl.py hws.getNodes
105 [node1] out: <twisted.python.failure.Failure <class '
    xmlrpclib.Fault'>>
106 [node1] out:
107
108 [node1] Executing task 'step2_start_hwd_client'
109 Use capictl to start hwdc on node1
110 [node1] run: python capictl.py supervisor.
    startProcess hwdc
111 [node1] out: True
112 [node1] out:
113
114
115 Clusterapi process tree:
116 [node1] run: pstree -la 'supervisorctl pid'
117 [node1] out: supervisord /home/vagrant/.virtualenvs/
    capi/bin/supervisord -c supervisord_node1.conf
118 [node1] out: python clusterapi/clusterapi.py
119 [node1] out: {python}
120 [node1] out: twistd /home/vagrant/.virtualenvs/capi
    /bin/twistd --nodaemon --pidfile=hws_node1.pid
    hwdserver --database=sqlite:///hwdiscover.sqlite
121 [node1] out: twistd /home/vagrant/.virtualenvs/capi
    /bin/twistd --nodaemon --pidfile=hwdc_node1.pid
    hwdclient
122 [node1] out: lsb_release -Es /usr/bin/
    lsb_release -r
123 [node1] out: (swapon)
124 [node1] out: {twistd}
125 [node1] out:
126
127 [node1] run: supervisorctl status hwdc
```

G. Acceptance test

```
128 [node1] out: hwdc RUNNING
      pid 26638, uptime 0:00:08
129 [node1] out:
130
131 [node1] run: python capictl.py listSubHandlers
132 [node1] out: ['node1', 'hwds', 'supervisor', 'system',
      , 'hwdc']
133 [node1] out:
134
135 [node1] run: python capictl.py listSubHandlerMethods
      hwdc
136 [node1] out: ['hwdc.test',
137 [node1] out: 'hwdc.update',
138 [node1] out: 'system.methodHelp',
139 [node1] out: 'system.listMethods',
140 [node1] out: 'system.methodSignature']
141 [node1] out:
142
143 [node1] run: python capictl.py hwds.getNodes
144 [node1] out: '["node1.local"]'
145 [node1] out:
146
147 [node1] Executing task 'step3_start_nge'
148 [node1] run: python capictl.py supervisor.
      startProcess nge
149 [node1] out: True
150 [node1] out:
151
152 [node2] Executing task 'step4_start_second_capi'
153
154 Start supervisord with the clusterapi configuration.
155 [node2] run: supervisord -c supervisord_node2.conf
156
157 Done.
158 Disconnecting from vagrant@127.0.0.1:2222... done.
159 Disconnecting from vagrant@127.0.0.1:2200... done.
```

Bibliography

- [1] Sterling T 2001 *International Journal of High Performance Computing Applications* **15** 92–101 ISSN 1094-3420 URL <http://hpc.sagepub.com/cgi/doi/10.1177/109434200101500202>
- [2] 2011 Architecture Share Over Time URL <http://www.top500.org/overtime/list/38/archtype>
- [3] Association for computing machinery URL <http://www.acm.org/>
- [4] SIGSOFT homepage URL <http://www.sigsoft.org/>
- [5] IEEE xplore: Computing in science & engineering URL <http://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=5992>
- [6] IEEE xplore - conference table of contents URL <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5054535>
- [7] International journal of high performance computing applications URL <http://hpc.sagepub.com/>
- [8] The SC conference series URL <http://www.supercomp.org/>
- [9] Kelly D 2007 *IEEE Software* **24** 120–119 ISSN 0740-7459
- [10] Faulk S, Loh E, Vanter M L V D, Squires S and Votta L G 2009 *Computing in Science Engineering* **11** 30–39 ISSN 1521-9615
- [11] Glass R L and Vessey I 1998 Focusing on the application domain: everyone agrees it's vital, but who's doing anything about it? , *Proceedings of the Thirty-First Hawaii International Conference on System Sciences, 1998* vol 3 pp 187–196 vol.3
- [12] Basili V, Carver J, Cruzes D, Hochstein L, Hollingsworth J, Shull F and Zelkowitz M 2008 *IEEE Software* **25** 29–36 ISSN 0740-7459
- [13] Schmidberger M and Brugge B 2012 Need of software engineering methods for high performance computing applications *2012 11th International Symposium on Parallel and Distributed Computing (ISPDC)* pp 40–46
- [14] Brooks F P 1995 *Mythical Man-Month, The: Essays on Software Engineering, Anniversary Edition, 2nd Edition* 2nd ed (Addison-Wesley Professional.) ISBN 0-201-83595-9, 978-0-201-83595-3 URL <http://www.informit.com/store/mythical-man-month-essays-on-software-engineering-anniversary-9780201835953>

Bibliography

- [15] Glass R L 2004 *Commun. ACM* **47** 19–21 ISSN 0001-0782 URL <http://doi.acm.org/10.1145/986213.986228>
- [16] Glass R L and Vessey I 2011 *IEEE Software* **28** 14–15 ISSN 0740-7459
- [17] 2010 Greg wilson - what we actually know about software development, and why we believe it's true URL <http://vimeo.com/9270320>
- [18] Wilson G 2010 Bits of evidence URL <http://www.slideshare.net/gwilson/bits-of-evidence-2338367>
- [19] Carminati F 2012 *Journal of Physics: Conference Series* **396** 052018 ISSN 1742-6596 URL <http://iopscience.iop.org/1742-6596/396/5/052018>
- [20] Wilson G 2008 *Computing in Science Engineering* **10** 5–6 ISSN 1521-9615
- [21] Wilson G 2006 *Computing in Science Engineering* **8** 66–69 ISSN 1521-9615
- [22] Fabjan C W, Jirdén L, Lindstruth V, Riccati L, Röhrich D, de Vyvre P V, Baillie O V and de Groot H 2004 *ALICE trigger data-acquisition high-level trigger and control system: Technical Design Report* Technical Design Report ALICE (Geneva: CERN)
- [23] Foster I and Kesselman C 2003 *The Grid 2: Blueprint for a New Computing Infrastructure* The Elsevier Series in Grid Computing (Elsevier Science) ISBN 9780080521534 URL <http://books.google.no/books?id=015gm6o3vrMC>
- [24] 2011 Setup of the ALICE detector system URL <http://aliceinfo.cern.ch/Public/Objects/Chapter2/ALICE-SetUp-NewSimple.jpg>
- [25] Brock I and Schörner-Sadenius T 2011 *Physics at the Terascale* (John Wiley & Sons) ISBN 9783527634972
- [26] Collaboration T A 2008 *J. Instrum.* **3** S08002. 259 p also published by CERN Geneva in 2010
- [27] Collaboration T A 2008 *J. Instrum.* **3** S08002. 259 p also published by CERN Geneva in 2010
- [28] Alme J, Andres Y, Appelshäuser H, Bablok S, Bialas N, Bolgen R, Bonnes U, Bramm R, Braun-Munzinger P, Campagnolo R, Christiansen P, Dobrin A, Engster C, Fehlker D, Foka Y, Frankenfeld U, Gaardhøje J J, Garabatos C, Glässel P, Gonzalez Gutierrez C, Gros P, Gustafsson H A, Helstrup H, Hoch M, Ivanov M, Janik R, Junique A, Kalweit A, Keidel R, Kniege S, Kowalski M, Larsen D T, Lesenechal Y, Lenoir P, Lindegaard N, Lippmann C, Mager M, Mast M, Matyja A, Munkejord M, Musa L, Nielsen B S, Nikolic V, Oeschler H, Olsen E K, Oskarsson A, Osterman L, Pikna M, Rehman A, Renault G, Renfordt R, Rossegger S, Röhrich D, Røed K, Richter M, Rueschmann G, Rybicki

Bibliography

- A, Sann H, Schmidt H R, Siska M, Sitár B, Soegaard C, Soltveit H K, Soyk D, Stachel J, Stelzer H, Stenlund E, Stock R, Strmeň P, Szarka I, Ullaland K, Vranic D, Veenhof R, Westergaard J, Wiechula J and Windelband B 2010 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **622** 316–367 ISSN 0168-9002 URL <http://www.sciencedirect.com/science/article/pii/S0168900210008910>
- [29] 2011 CERN Advanced STORage manager URL <http://castor.web.cern.ch>
- [30] Franek B and Gaspar C 1997 SMI++-object oriented framework for designing control systems *Nuclear Science Symposium, 1997. IEEE* pp 92–96 vol.1 ISSN 1082-3654
- [31] Holme O, González-Berges M, Golonka P and Schmeling S 2005 The JCOP Framework Tech. Rep. CERN-OPEN-2005-027 CERN Geneva
- [32] 2012 PVSS URL <http://www.pvss.com>
- [33] Dobre C and Stratan C 2011 *CoRR* **abs/1106.5158**
- [34] Bagnasco S, Betev L, Buncic P, Carminati F, Cirstoiu C, Grigoras C, Hayrapetyan A, Harutyunyan A, Peters A J and Saiz P 2008 *J. Phys.: Conf. Ser.* **119** 062012
- [35] Legrand I, Newman H, Voicu R, Cirstoiu C, Grigoras C, Toarta M and Dobre C 2005
- [36] Elsen E and Haller J 2011 *Trigger Systems in High Energy Physics Experiments* (Wiley-VCH Verlag GmbH & Co. KGaA) pp 363–382 ISBN 9783527634965 URL <http://dx.doi.org/10.1002/9783527634965.ch18>
- [37] Cappello F 2009 *International Journal of High Performance Computing Applications* **23** 212–226 ISSN 1094-3420, 1741-2846 URL <http://hpc.sagepub.com/content/23/3/212>
- [38] Electronics racks | ALICE collaboration URL http://aliceinfo.cern.ch/Technical_Coordination/ElectronicsRacks.html
- [39] Thäder J 2012 Commissioning of the ALICE high-level trigger URL <https://cds.cern.ch/record/1561431>
- [40] Szostak A 2012 *Journal of Physics: Conference Series* **396** 012048 ISSN 1742-6596 URL <http://iopscience.iop.org/1742-6596/396/1/012048>
- [41] Cortese P, Carminati F, Fabjan C W, Riccati L and de Groot H 2005 *ALICE computing: Technical Design Report* Technical Design Report ALICE (Geneva: CERN) submitted on 15 Jun 2005
- [42] Alt T, Lindenstruth V and Tilsner H 2006 URL http://inis.iaea.org/Search/search.aspx?orig_q=RN:37102167

Bibliography

- [43] Rohr D 2012 ALICE TPC online tracker on GPUs for heavy-ion events *2012 13th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA)* pp 1–6
- [44] Kollegger T 2012 The ALICE high level trigger: The 2011 run experience *Real Time Conference (RT), 2012 18th IEEE-NPSS* pp 1–4
- [45] ALICE high-level trigger conceptual design URL <https://wiki.kip.uni-heidelberg.de/ti/HLT/images/c/cf/CDR-HLT-CERN-20021217.pdf>
- [46] Villalobos Baillie O, Riccati L, Rorich D, Jirdén L, Fabjan C W, Van de Vyvre P, de Groot H and Lindestruth V 2004 ALICE trigger data-acquisition high-level trigger and control system URL <http://cds.cern.ch/record/684651>
- [47] Richardson T, Stafford-Fraser Q, Wood K and Hopper A 1998 *IEEE Internet Computing* **2** 33–38 ISSN 1089-7801
- [48] Panse R E 2009 CHARM-Card: hardware based cluster control and management system URL <http://archiv.ub.uni-heidelberg.de/volltextserver/10013/>
- [49] 2014 Intelligent platform management interface page Version ID: 626893050 URL http://en.wikipedia.org/w/index.php?title=Intelligent_Platform_Management_Interface&oldid=626893050
- [50] Richter M 2009 *Development and Integration of on-line Data Analysis for the ALICE Experiment* Doctoral thesis The University of Bergen URL <http://bora.uib.no/handle/1956/3555>
- [51] Steinbeck T M 2004 *arXiv:cs/0404014* URL <http://arxiv.org/abs/cs/0404014>
- [52] Richter M 2012 *Journal of Physics: Conference Series* **396** 012043 ISSN 1742-6596 URL <http://iopscience.iop.org/1742-6596/396/1/012043>
- [53] Bablok S 2008 *Heterogeneous distributed calibration framework for the high level trigger in ALICE* (University of Bergen) ISBN 9788230807408 URL <https://bora.uib.no/handle/1956/3857>
- [54] Lara Martinez C E and Keschull U 2011 *The SysMES Framework: System Management for Networked Embedded Systems and Clusters*. *oai:cds.cern.ch:1454269* Ph.D. thesis Heidelberg U. presented 19 Jan 2012
- [55] ROOT | a data analysis framework <http://root.cern.ch/drupal/> URL <http://root.cern.ch/drupal/>
- [56] Steinbeck T M 2004 A Modular and Fault-Tolerant Data Transport Framework (*Preprint cs/0404014v1*) URL <http://arxiv.org/abs/cs/0404014v1>; <http://arxiv.org/pdf/cs/0404014v1>

Bibliography

- [57] Rubin G, Van de Vyvre P, Csató P, Dénes E, Kiss T, Meggyesi Z, Sulyán J, Vesztergombi G, Eged B, Gerencsér I, Novák I, Soós C, Tarján D, Telegdy A and Tóth N 1999 6 p
- [58] C Cheshkov R D and Steinbeck T M 2008 Identification of ddl links in alice data URL <https://edms.cern.ch/document/871996>
- [59] Divià R, Jovanovic P and Van de Vyvre P 2007 Data format over the ALICE DDL URL <https://cds.cern.ch/record/1027339>
- [60] Grosse-Oetringhaus J F, Zampolli C, Colla A, Carminati F and Collaboration t A 2010 *Journal of Physics: Conference Series* **219** 022010 ISSN 1742-6596 URL <http://iopscience.iop.org/1742-6596/219/2/022010>
- [61] 2012 OpenBSD URL <http://www.openbsd.org>
- [62] 2012 Gentoo URL <http://www.gentoo.org>
- [63] 2012 Debian URL <http://www.debian.org>
- [64] Middlelink J P R 2003 Bigphysarea kernel patch URL <http://www.polyware.nl/middelink/En/hob-v41.html#bigphysarea>
- [65] Engwer C 2002 Pci and shared memory interface URL <http://www.kip.uni-heidelberg.de/HLT/software/download/psi-0.8.2.tar.gz>
- [66] 2012 Environment Modules URL <http://modules.sourceforge.net>
- [67] Furlani J L and Osel P W 1996 Abstract Yourself With Modules pp 193–204 URL <http://dblp.uni-trier.de/rec/bibtex/conf/lisa/Furlani096>
- [68] 2011 DATE URL <https://ph-dep-aid.web.cern.ch/ph-dep-aid>
- [69] Carena W, Van de Vyvre P, Carena F, Chapeland S, Chibante Barroso V, Costa F, Denes E, Divia R, Fuchs U, Simonetti G, Soos C, Telesca A and von Haller B 2005 ALICE DAQ and ECS User's Guide Tech. Rep. ALICE-INT-2005-015. CERN-ALICE-INT-2005-015 CERN Geneva
- [70] Robertson A 2000 *Proc. 4th Annual Linux Showcase & Conferenc* **4** 20–20 URL http://static.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/robertson/robertson_html/
- [71] Reisner P and Ellenberg L 2005 Replicated storage with shared disk semantics *Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress), Germany* p 111–119 URL <http://www.drbd.org/fileadmin/drbd/publications/drbd8.pdf>
- [72] 2014 LAMP (software bundle) page Version ID: 613261149 URL [http://en.wikipedia.org/w/index.php?title=LAMP_\(software_bundle\)&oldid=613261149](http://en.wikipedia.org/w/index.php?title=LAMP_(software_bundle)&oldid=613261149)

Bibliography

- [73] Big design up front URL <http://c2.com/cgi/wiki?BigDesignUpFront>
- [74] 2012 Big design up front page Version ID: 522882580 URL http://en.wikipedia.org/w/index.php?title=Big_Design_Up_Front&oldid=522882580
- [75] Kepner J 2004 *International Journal of High Performance Computing Applications* **18** 505–516 ISSN 1094-3420, 1741-2846 URL <http://hpc.sagepub.com/content/18/4/505>
- [76] Bajko M, Dahlerup-Petersen K, Kirby G, Parma V, van Weelderen R, Koratzinos M, Mess K H, Todesco E, Claudet S, Lebrun P, Modena M, Strait J, Perin A, Feher S, Rijllart A, Siemko A, Montabonnet V, Flora R, Schmidt R, Bertinelli F, Rossi L, de Rijk G, Cruikshank P, Limon P, Strubin P, Wolf R, Verweij A, Fessia P, Tock J P, Garion C, Tavian L, Jimenez J M, Thiesen H, Le Naour S, Walckiers L, Catalan Lasherias N, Veness R, Denz R and Nunes R 2009 URL <https://inspirehep.net/record/824814?ln=en>
- [77] Aamodt K *et al.* (ALICE Collaboration) 2010 *Eur.Phys.J.* **C65** 111–125 (*Preprint* 0911.5430) URL <http://dx.doi.org/10.1140/epjc/s10052-009-1227-4>
- [78] ALICE enters new territory in heavy-ion collisions - CERN courier the LHC dishes up hot dense matter at higher energies than ever before. URL <http://cerncourier.com/cws/article/cern/46055>
- [79] Ram D, Breitner T and Szostak A 2012 *Journal of Physics: Conference Series* **396** 012042 ISSN 1742-6596 URL <http://iopscience.iop.org/1742-6596/396/1/012042>
- [80] Erdal H A, Richther M, Szostak A and Toia A 2012 *Journal of Physics: Conference Series* **396** 012019 ISSN 1742-6596 URL <http://iopscience.iop.org/1742-6596/396/1/012019>
- [81] Carena F, Carena W, Chapeland S, Barroso V, Costa F, Dé andnes E, Divià and R, Fuchs U, Simonetti G, Soó ands C, Telesca A, Vande Vyvre P and Von Haller B 2010 The ALICE Electronic Logbook *Real Time Conference (RT), 2010 17th IEEE-NPSS* pp 1–5
- [82] Gerhards R 2009 The Syslog Protocol RFC 5424 (Proposed Standard) URL <http://www.ietf.org/rfc/rfc5424.txt>
- [83] CEE — 1.0-beta1 CEE overview <http://cee.mitre.org/language/1.0-beta1/overview.html> URL <http://cee.mitre.org/language/1.0-beta1/overview.html>
- [84] Boehm B W 1981 *Software engineering economics* (Prentice-Hall) ISBN 9780138221225
- [85] Boehm B W 1984 *IEEE Trans. Software Eng.* **10** 4–21 URL <http://dblp.uni-trier.de/db/journals/tse/tse10.html#Boehm84>

Bibliography

- [86] Albrecht A and Gaffney JE J 1983 *IEEE Transactions on Software Engineering* **SE-9** 639 – 648 ISSN 0098-5589
- [87] El Emam K, Benlarbi S, Goel N and Rai S 2001 *IEEE Transactions on Software Engineering* **27** 630–650 ISSN 0098-5589
- [88] Kepner J 2004 *International Journal of High Performance Computing Applications* **18** 393–397 ISSN 1094-3420, 1741-2846 URL <http://hpc.sagepub.com/content/18/4/393>
- [89] Kemerer C F 1987 *Commun. ACM* **30** 416–429 ISSN 0001-0782 URL <http://doi.acm.org/10.1145/22899.22906>
- [90] Wheeler D 2009 SLOCcount URL <http://www.dwheeler.com/sloccount>
- [91] GitStats - git history statistics generator <http://gitstats.sourceforge.net/> URL <http://gitstats.sourceforge.net/>
- [92] Kruchten P 2003 *The Rational Unified Process: An Introduction* 3rd ed (Addison-Wesley Professional) ISBN 0321197704
- [93] File:Development-iterative.gif URL <http://en.wikipedia.org/wiki/File:Development-iterative.gif>
- [94] File:UnifiedProcessProjectProfile20060708.png URL <http://en.wikipedia.org/wiki/File:UnifiedProcessProjectProfile20060708.png>
- [95] 2012 Top 500 Supercomputer Sites URL <http://www.top500.org>
- [96] 2012 HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers URL <http://www.netlib.org/benchmark/hpl>
- [97] EPA report on server and data center energy efficiency : ENERGY STAR http://www.energystar.gov/index.cfm?c=prod_development.server_efficiency_study URL http://www.energystar.gov/index.cfm?c=prod_development.server_efficiency_study
- [98] 2012 The Green500 List :: Environmentally Responsible Supercomputing URL <http://www.green500.org>
- [99] Cunningham W 1992 The WyCash portfolio management system *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)* OOPSLA '92 (New York, NY, USA: ACM) p 29–30 ISBN 0-89791-610-7 URL <http://doi.acm.org/10.1145/157709.157715>
- [100] Code coverage URL <http://c2.com/cgi/wiki?CodeCoverage>
- [101] Production code vs unit tests ratio URL <http://c2.com/cgi/wiki?ProductionCodeVsUnitTestsRatio>

Bibliography

- [102] Martin R C 2002 *Agile Software Development, Principles, Patterns, and Practices* 2nd ed (Prentice Hall) ISBN 0135974445, 9780135974445
- [103] Fowler M, Beck K, Brant J, Opdyke W and Roberts D 2012 *Refactoring: Improving the Design of Existing Code* (Addison-Wesley) ISBN 9780133065268
- [104] Simian - similarity analyser | duplicate code detection for the enterprise | overview URL <http://www.harukizaemon.com/simian/>
- [105] the archive of CCFinder official site URL <http://www.ccfinder.net/>
- [106] Roy C K and Cordy J R 2007 *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY* 115
- [107] Kephart J O and Chess D M 2003 *Computer* **36** 41–50 ISSN 0018-9162 URL <http://dx.doi.org/10.1109/MC.2003.1160055>
- [108] Horn 2001 *IBM Corporation (October 15, 2001). Available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf*
- [109] Prechelt L 2000 *Computer* **33** 23–29 ISSN 0018-9162
- [110] Function point languages table | QSM SLIM-Estimate URL <http://www.qsm.com/resources/function-point-languages-table>
- [111] Paulson L D 2007 *Computer* **40** 12–15 ISSN 0018-9162
- [112] Lloyd J 1994 Practical advantages of declarative programming
- [113] Dijkstra E W 1982 *Selected writings on computing: a personal perspective* (New York, NY, USA: Springer-Verlag New York, Inc.) ISBN 0-387-90652-5
- [114] Introduction to qt quick | documentation | qt project <http://qt-project.org/doc/qt-4.8/qml-intro.html> URL <http://qt-project.org/doc/qt-4.8/qml-intro.html>
- [115] 2011 Cross-platform application and UI framework URL <http://qt.nokia.com>
- [116] QtQuick 5.0: Qt quick scene graph | documentation | qt project <http://qt-project.org/doc/qt-5.0/qtquick/qtquick-visualcanvas-scenegraph.html> URL <http://qt-project.org/doc/qt-5.0/qtquick/qtquick-visualcanvas-scenegraph.html>
- [117] Declarative — SQLAlchemy 0.8 documentation http://docs.sqlalchemy.org/en/rel_0_8/orm/extensions/declarative.html URL http://docs.sqlalchemy.org/en/rel_0_8/orm/extensions/declarative.html
- [118] 2008 *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multi-core programming* (New York, NY, USA: ACM) ISBN 978-1-60558-417-1 553091

Bibliography

- [119] Lerner J and Tirole J 2005 *Journal of Economic Perspectives* **19** 99–120 ISSN 0895-3309 URL <http://www.aeaweb.org/articles.php?doi=10.1257/0895330054048678>
- [120] Mission | Open Source Initiative <http://www.opensource.org/> URL <http://www.opensource.org>
- [121] Free Software Foundation — Free Software Foundation — working together for free software <http://www.fsf.org/> URL <http://www.fsf.org>
- [122] Lerner J and Tirole J 2004 *National Bureau of Economic Research Working Paper Series No. 10956* URL <http://www.nber.org/papers/w10956>
- [123] Haefliger S, Von Krogh G and Spaeth S 2008 *Management Science* **54** 180–193 ISSN 0025-1909, 1526-5501 URL <http://mansci.journal.informs.org/content/54/1/180>
- [124] Coverity scan URL <http://scan.coverity.com/>
- [125] Magoules F 2009 *Fundamentals of Grid Computing: Theory, Algorithms and Technologies* 1st ed (Boca Raton: Chapman and Hall/CRC) ISBN 9781439803677
- [126] Home » OpenStack open source cloud computing software URL <https://www.openstack.org/>
- [127] Rackspace joins CERN openlab as a contributor URL <http://openlab.web.cern.ch/news/rackspace-joins-cern-openlab-contributor>
- [128] Coupling and cohesion URL <http://c2.com/cgi/wiki?CouplingAndCohesion>
- [129] 2014 Representational state transfer page Version ID: 627842683 URL http://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=627842683
- [130] Fielding R T 2000 *Architectural Styles and the Design of Network-based Software Architectures* Ph.D. thesis University of California, Irvine URL <http://www.ics.uci.edu/fielding/pubs/dissertation/top.htm>
- [131] Wikipedia 2011 Representational state transfer — Wikipedia, The Free Encyclopedia [Online; accessed 8-December-2011] URL http://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=462445552
- [132] Haerder T and Reuter A 1983 *ACM Comput. Surv.* **15** 287–317 ISSN 0360-0300 URL <http://doi.acm.org/10.1145/289.291>
- [133] Kegel D The C10K problem URL <http://www.kegel.com/c10k.html>
- [134] C10M: the C10M problem URL <http://c10m.robertgraham.com/p/manifesto.html>

Bibliography

- [135] High scalability - high scalability - the secret to 10 million concurrent connections -the kernel is the problem, not the solution URL <http://highscalability.com/blog/2013/5/13/the-secret-to-10-million-concurrent-connections-the-kernel-i.html>
- [136] O'Grady S *The New Kingmakers* URL <http://shop.oreilly.com/product/0636920028185.do>
- [137] 2011 Relational Persistence for Java and .NET URL <http://www.hibernate.org>
- [138] 2011 The Python SQL Toolkit and Object Relational Mapper URL <http://www.sqlalchemy.org>
- [139] 2011 Twisted event-driven network engine URL <http://twistedmatrix.com/trac>
- [140] 2011 Introduction to Perspective Broker URL <http://twistedmatrix.com/documents/current/core/howto/pb-intro.html>
- [141] Distributed computing made simple - zeromq URL <http://zeromq.org/>
- [142] protobuf - protocol buffers - google's data interchange format - google project hosting <http://code.google.com/p/protobuf/> URL <http://code.google.com/p/protobuf/>
- [143] Apache thrift <http://thrift.apache.org/about/> URL <http://thrift.apache.org/about/>
- [144] BSON - binary JSON <http://bsonspec.org/> URL <http://bsonspec.org/>
- [145] MongoDB <http://www.mongodb.org/> URL <http://www.mongodb.org/>
- [146] MessagePack: it's like JSON. but fast and small. <http://msgpack.org/> URL <http://msgpack.org/>
- [147] Krochmal M and Cheshire S DNS-based service discovery URL <http://tools.ietf.org/html/rfc6763>
- [148] Guttman E, Perkins C, Veizades J and Day M 1999 Service Location Protocol, Version 2 RFC 2608 (Proposed Standard) updated by RFC 3224 URL <http://www.ietf.org/rfc/rfc2608.txt>
- [149] 2011 Avahi URL <http://avahi.org>
- [150] 2011 OpenSLP URL <http://www.openslp.org>
- [151] 2011 WBEM URL <http://www.dmtf.org/standards/wbem>
- [152] DMTF 2011 WBEM Discovery Using the Service Location Protocol (SLP) Tech. rep. URL http://www.dmtf.org/sites/default/files/standards/documents/DSP0205_1.0.0.pdf

Bibliography

- [153] 2011 Common Information Model URL <http://dmtf.org/standards/cim>
- [154] 2011 Managed Object Format URL <http://www.dmtf.org/education/mof>
- [155] SourceForge.net: sblim http://sourceforge.net/apps/mediawiki/sblim/index.php?title=Main_Page
URL http://sourceforge.net/apps/mediawiki/sblim/index.php?title=Main_Page
- [156] OpenPegasus <https://collaboration.opengroup.org/pegasus/> URL <https://collaboration.opengroup.org/pegasus/>
- [157] GitHub · build software better, together. URL <https://github.com/>
- [158] Free source code hosting for git and mercurial by bitbucket URL <https://bitbucket.org/>
- [159] 2012 Fabric documentation URL <http://docs.fabfile.org>
- [160] Goodger D and van Rossum G 2012 Docstring Conventions URL <http://www.python.org/dev/peps/pep-0257>
- [161] Overview — Sphinx 1.1.3 documentation <http://sphinx.pocoo.org/> URL <http://sphinx.pocoo.org>
- [162] Epydoc <http://epydoc.sourceforge.net/> URL <http://epydoc.sourceforge.net>
- [163] Calcote J 2010 *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool* (No Starch Pr)
- [164] 2011 CMake URL <http://www.cmake.org>
- [165] Hoffman B, Cole D and Vines J 2009 Software Process for Rapid Development of HPC Software Using CMake *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009* pp 378–382
- [166] 2011 Scientific Linux CERN URL <http://linux.web.cern.ch/linux/scientific.shtml>
- [167] 2012 CMake RPATH handling URL http://www.cmake.org/Wiki/CMake_RPATH_handling
- [168] 2012 Exit Status - The GNU C Library URL http://www.gnu.org/software/libc/manual/html_node/Exit-Status.html
- [169] 2012 Valgrind URL <http://valgrind.org>
- [170] 2012 Gcov - Using the GNU Compiler Collection URL <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>
- [171] 2011 CDash URL <http://www.cdash.org>

Bibliography

- [172] Welcome to jenkins CI! | jenkins CI URL <http://jenkins-ci.org/>
- [173] CruiseControl home URL <http://cruisecontrol.sourceforge.net/>
- [174] Buildbot URL <http://buildbot.net/>
- [175] ContinuousDelivery URL <http://martinfowler.com/bliki/ContinuousDelivery.html>
- [176] 2012 The Debian GNU/Linux FAQ - Basics of the Debian package management system URL <http://www.debian.org/doc/manuals/debian-faq/ch-pkg-basics.html>
- [177] 2012 rpm - Trac URL <http://www.rpm.org>
- [178] 2012 AptPreferences - Debian Wiki URL <http://wiki.debian.org/AptPreferences#Pinning>
- [179] 2012 APT - Debian Wiki URL <http://wiki.debian.org/Apt>
- [180] Fagan M E 1976 *IBM Systems Journal* **15** 182–211 ISSN 0018-8670
- [181] Cohen J A 2006 *Best Kept Secrets of Peer Code Review* (Printing Systems) ISBN 9781599160672
- [182] Benington H D 1983 *Annals of the History of Computing* **5** 350–361 ISSN 0164-1239
- [183] Beck K 2003 *Test-Driven Development: By Example*
- [184] Fowler M, Beck K, Brant J, Opdyke W and Roberts D 1999 *Refactoring: Improving the Design of Existing Code* 1st ed (Addison-Wesley Professional. Part of the Addison-Wesley Object Technology Series series.) ISBN 0-201-48567-2, 978-0-201-48567-7 URL <http://www.informit.com/store/refactoring-improving-the-design-of-existing-code-9780201485677>
- [185] Kerievsky J 2004 *Refactoring to Patterns* 1st ed (Addison-Wesley Professional. Part of the Addison-Wesley Signature Series (Fowler) series.) ISBN 0-321-21335-1, 978-0-321-21335-8 URL <http://www.informit.com/store/refactoring-to-patterns-9780321213358>
- [186] PEP 287 – reStructuredText docstring format URL <http://www.python.org/dev/peps/pep-0287/>
- [187] 25.2. doctest — Test interactive Python examples — Python v2.7.3 documentation <http://docs.python.org/library/doctest.html> URL <http://docs.python.org/library/doctest.html>
- [188] Knuth D E 1984 *The Computer Journal* **27** 97–111 ISSN 0010-4620, 1460-2067 URL <http://comjnl.oxfordjournals.org/content/27/2/97>

Bibliography

- [189] Agile Best Practice: Executable Specifications
<http://www.agilemodeling.com/essays/executableSpecifications.htm> URL
<http://www.agilemodeling.com/essays/executableSpecifications.htm>
- [190] Cunningham W Fit: Framework for Integrated Test <http://fit.c2.com/> URL <http://fit.c2.com>
- [191] FitNesse - The fully integrated standalone wiki, and acceptance testing framework.
<http://fitnesse.org/> URL <http://fitnesse.org>
- [192] Raggett D, Hors A L, Jacobs I *et al.* 1999 *W3C recommendation 24*
- [193] North D 2007 Available in: <http://dannorth.net/introducing-bdd>
- [194] Cucumber - Making BDD fun <http://cukes.info/> URL <http://cukes.info>
- [195] Beck K 2005 *Extreme Programming Explained* (Boston: Addison-Wesley) ISBN 9780321278654
- [196] Nordberg M I 2003 *Software, IEEE* **20** 26-33 ISSN 0740-7459
- [197] Harrington D, Presuhn R and Wijnen B 2002 An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks RFC 3411 (Standard) updated by RFCs 5343, 5590 URL <http://www.ietf.org/rfc/rfc3411.txt>
- [198] 2011 splunk URL <http://www.splunk.com>
- [199] 2011 Nagios URL <http://www.nagios.org>
- [200] 2011 Zenoss URL <http://www.zenoss.com>
- [201] 2011 Ganglia URL <http://ganglia.sourceforge.net>
- [202] 2011 Lemon URL <http://lemon.web.cern.ch/lemon/index.shtml>
- [203] Munin URL <http://munin-monitoring.org/>
- [204] Rainer's blog: Announcing project lumberjack
<http://blog.gerhards.net/2012/02/announcing-project-lumberjack.html> URL
<http://blog.gerhards.net/2012/02/announcing-project-lumberjack.html>
- [205] Project lumberjack to improve linux logging « bazsi's blog
<http://bazsi.blogs.balabit.com/2012/02/project-lumberjack-to-improve-linux-logging/> URL
<http://bazsi.blogs.balabit.com/2012/02/project-lumberjack-to-improve-linux-logging/>
- [206] Common event expression: CEE, a standard log language for event interoperability in electronic systems <http://cee.mitre.org/> URL <http://cee.mitre.org/>

Bibliography

- [207] statsd URL <https://github.com/etsy/statsd>
- [208] Start page – collectd – the system statistics collection daemon URL <https://collectd.org/>
- [209] Graphite documentation — graphite 0.10.0 documentation URL <http://graphite.readthedocs.org/en/latest/>
- [210] RRDtool - about RRDtool URL <http://oss.oetiker.ch/rrdtool/>
- [211] Cownie J, Moore S *et al.* 2000 Portable OpenMP debugging with totalview *Proceedings of the Second European Workshop on OpenMP (EWOMP 2000)*, Citeseer
- [212] Mackinnon T, Freeman S and Craig P 2001 *Endo-testing: unit testing with mock objects* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.) pp 287–301 ISBN 0-201-71040-4 URL <http://portal.acm.org/citation.cfm?id=377517.377534>
- [213] PyPy - welcome to PyPy URL <http://pypy.org/>
- [214] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn D and Smith K 2011 *Computing in Science Engineering* **13** 31–39 ISSN 1521-9615
- [215] WBEM solutions | home URL <http://www.ws-inc.com/>
- [216] SourceForge.net: pywbem URL http://sourceforge.net/apps/mediawiki/pywbem/index.php?title=Main_Page
- [217] 2011 Bright Cluster Manager URL <http://www.clustervision.com/bright-cluster-manager>
- [218] 2011 Pacemaker URL <http://www.clusterlabs.org>
- [219] Dake S, Caulfield C and Beekhof A The Corosync Cluster Engine *Linux Symposium* p 85
- [220] Yoo A, Jette M and Grondona M 2003 SLURM: Simple Linux Utility for Resource Management *Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science vol 2862)* ed Feitelson D, Rudolph L and Schwiegelshohn U (Springer Berlin / Heidelberg) pp 44–60 ISBN 978-3-540-20405-3 URL <http://www.springerlink.com/content/c4pgx63utdajtuwn/abstract>
- [221] Smith B C 1982 *Procedural reflection in programming languages* Thesis Massachusetts Institute of Technology thesis (Ph.D.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1982. URL <http://dspace.mit.edu/handle/1721.1/15961>
- [222] 2011 ZeroMQ URL <http://www.zeromq.org>

Bibliography

- [223] 2011 RabbitMQ URL <http://www.rabbitmq.com>
- [224] Supervisor: A process control system — supervisor 3.1a1-dev documentation URL <http://supervisord.org/>
- [225] Hermann M | 2010 *Object-Relational Mapping for the Common Information Model* URL http://sfxeu09.hosted.exlibrisgroup.com/sfx_ubb?url_ver=Z39.88-2004&rft_id=info%3Aid%2FIEEE.org%3AXPLORE&url_ctx_fmt=info%3Aofi%2Ffmt%3Akev%3Amtx%3Actx&rft_val_fmt=info%3Aofi%2Ffmt%3Akev%3Amtx%3Ajournal&rft.jtitle=Object-Relational%20Mapping%20for%20the%20Common%20Information%20Model&rft.date=2010&rft.au=M.%20Hermann&rft.aulast=M.%20Hermann
- [226] Haaland O, Hermann M, Ulrich J, Lara C, Rohrich D and Keschull U 2011 Realization of inventory databases and object relational mapping for the common information model *Systems and Virtualization Management (SVM), 2011 5th International DMTF Academic Alliance Workshop on* pp 1–8
- [227] *Date: 2009-05-01 Version: 2.5.0 5 Common Information Model (CIM) Infrastructure*
- [228] Mapping class inheritance hierarchies — SQLAlchemy 0.9 documentation URL http://docs.sqlalchemy.org/en/rel_0_9/orm/inheritance.html
- [229] Attribute overriding in mixin inheritance hierarchy – google grupper URL <https://groups.google.com/forum/?fromgroups#!topic/sqlalchemy/jVyRc2xmEQ>
- [230] 0.6 changelog — SQLAlchemy 0.9 documentation URL http://docs.sqlalchemy.org/en/latest/changelog/changelog_06.html#change-b0075b372f99003ca5b69e678f306150
- [231] Love R 2005 *Linux Journal* **2005** 8
- [232] 5. built-in types — python v2.7.6 documentation URL <http://docs.python.org/2/library/stdtypes.html#string-formatting-operations>
- [233] Horn 2001 *IBM Corporation (October 15, 2001). Available at* http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf
- [234] Kephart J O and Chess D M 2003 *Computer* **36** 41–50 ISSN 0018-9162 URL <http://dx.doi.org/10.1109/MC.2003.1160055>
- [235] 2012 nose - is nicer testing for python URL <http://readthedocs.org/docs/nose/en/latest>
- [236] 2012 coverage.py URL <http://nedbatchelder.com/code/coverage>
- [237] 2012 pylint URL <http://www.logilab.org/857>

Bibliography

- [238] 2012 TwistedTrial URL <http://twistedmatrix.com/trac/wiki/TwistedTrial>
- [239] Vagrant URL <http://www.vagrantup.com/>
- [240] ghost.py URL <http://jeanphix.me/Ghost.py/>
- [241] Welcome to fabric! — fabric documentation URL <http://www.fabfile.org/>
- [242] 2014 Cruft page Version ID: 635549916 URL <http://en.wikipedia.org/w/index.php?title=Cruft&oldid=635549916>
- [243] 2011 Security Considerations URL <http://avahi.org/wiki/SecurityConsiderations>
- [244] 2011 MySQL URL <http://www.mysql.com>
- [245] 2011 Python URL <http://www.python.org>
- [246] 2011 matplotlib URL <http://matplotlib.sourceforge.net>

