Master thesis

in informatics – Software development

Improvements and extensions to the configuration tools of the TaskManager HLT software in the ALICE experiment at CERN

Øystein Senneset Haaland

University of Bergen
November 2007

Abstract

This thesis concerns work done to improve and extend software that is used to configure nodes and trigger software in High-Level Trigger (HLT) under the ALICE experiment at CERN. HLT is a large computer cluster that will be used to reduce data from the ALICE experiment to a an amount manageable by storage systems. To accomplish this efficiently the processing is distributed throughout the cluster and the configuration tools are used to decide how this distribution should be done. The configuration tools are composed of a few Python programs and templates, which generates configuration files for all participating nodes by taking input from a single Extensible Markup Language (XML) file describing the entire setup.

While optimization and general improvement is one part of this thesis, the emphasis will be on extending the software to incorporate functionality for creating a more structured control hierarchy in the HLT data processing framework.

This thesis will present the tasks to be resolved, discuss possible solutions and describe the development process.

Preface

The work with this thesis has been both exciting and challenging. It has been a journey of hard work and many leassons learned. Working in the setting of a large, world wide collaboration as the ALICE experiment, has given experience with international work that can hardly be matched elsewhere for a master student. For a student with a background in informatics, getting to know the world of physics from the inside, has given new perspectives to how the world is viewed. All in all, it has been an extremely rewarding experience.

I would like to thank my supervisor Håvard Helstrup for prompt and comprehensive feedback all along the process, and for tirelessly reading and commenting drafts of my thesis. Kristin Fanebust Hetland, also at Høgskulen i Bergen, has done a great job of following-up my progress and providing useful advice.

The assignment for this thesis has been given by Institutt for Fysikk og Teknologi on behalf of the ALICE experiment at CERN. The institute has therefore been where i have had my work space. At the institute i would like to thank Dieter Roehrich for giving me the opportunity to work abroad at CERN, in Heidelberg and Paderborn. I would also like to thank those with whom i have shared office – Are, Knut, Kyrre, Torstein and Øystein – and from our group – Dag, Gaute, Kenneth, Matthias and Sebastian – for inspiration, being helpful and for adding a social aspect to the experience.

Further, i would like to thank our colleagues and HLT collaborators at Kirchhoff Institute of Physics in Heidelberg for invaluable impulses. In particular Timm M. Steinbeck for proposing such an exciting assignment for this thesis and for providing valuable input.

Thanks also to Lisa for proofreading my English and student advisor Ida for helping out with all practical problems.

Finally, I would like to thank my family, my parents in particular, for selflessly encourageing me to go my own ways and for supporting my efforts.

Bergen, November 2007
Øystein S. Haaland

Contents

List of Figures

Listings

List of Tables

1. Introduction

CERN is the European Organization for Nuclear Research. It employs the world's largest particle physics laboratory to enable a global community of scientists to explore the nature of our smallest particles. At its location on the border between Switzerland and France, there is continuous activity going on, either preparing or conducting experiments. The latest project, Large Hadron Collider (LHC), is bound for startup sometime in 2008 after having been under construction for almost two decades. One of the projects that is part of LHC is ALICE. Like the other experiments, ALICE produces a lot of data. To the extent that it is necessary to limit the data so it does not exceed the capabilities of the storage facilities. The mechanism to achieve this is the ALICE HLT, which helps select interesting data and compress it sufficiently for storage.

The HLT is implemented as a big cluster of computers that work together to analyse data in realtime. To be able to do this efficiently, software is set up in a hierarchy of distributed analysis processes that communicates over the local network. Setting up this complex structure of processes requires computer software. Configuration tools have therefore been developed to create configurations and startup scripts for all the participating machines according to specifications given in a master chain configuration.

This thesis discusses extending the functionality of these tools and in various ways improve upon them. More specifically to finalize the implementation of a concept that will help improve reliability in an analysis chain and to introduce software engineering concepts where it makes sense.

This chapter will be a short introduction to CERN, the ALICE experiment, HLT and the tasks to be solved.

1.1. Physics background

LHC is all about fundamental research and its general purpose is to improve our understanding of the universe. The theories, as they exist today, do not explain all of the observations that we make about nature. The prevailing theory, the Standard Model, leaves many questions unanswered. To scientists, interesting questions are for instance: why do elementary particles have mass and why do the mass differ among the particles[7]. The answer to such questions may lie in other, not yet discovered or verified particles. LHC will try to create conditions where these rare and volatile particles can be made detectable.

These particles are believed to only exist under conditions similar to those that immediately followed the birth of the universe – with large concentrations of energy,

and only for a very short stretch of time. Before long, their nature is changed – transformed (decayed) into a different particle – due to interactions with other particles or changes in its surroundings. In order to create such huge density of energy, very large machines are used to accelerate larger particles up to speeds close to the speed of light. As the speed increases, so does the energy contained in the travelling particles. When reaching a sufficiently high speed, the particles are collided head on, resulting in a shower of smaller particles. It is within this shower of mostly well known particles that the different experiments will try to discover new and interesting things. Ironically, to get a glimpse of these, our smallest particles, a huge machine is needed.

1.1.1. CERN

CERN (the European Organization for Nuclear Research), one of the first joint ventures in Europe, was founded in 1954. Here, close to Geneva on the border between France and Switzerland, scientists from all over the globe unite to find the building blocks of matter and the forces that hold them together[7]. It is the largest particle physics center in the world and employs about 3000 people from 20 member states. The entire CERN community is far more extensive, and includes 10000 scientists from 80 countries[7], all working together to construct the largest particle accelerator – and accompanying experiments – the world has ever seen: LHC (Large Hadron Collider).

The previous accelerator, Large Electron-Positron Collider (LEP), finished its operations in 2000 and was then dismantled to make place for the new LHC. LEP helped find the mass of Z and W boson, amongst many other things.

1.1.2. LHC - Large Hadron Collider

The Large Hadron Collider, is as it name implies, a large particle accelerator and collider. It has been under construction since 1984 and is first now entering the finalization stage, being prepared for startup in 2008. The purpose of the LHC is to study the behaviour of our smallest particles and from this information derive their characteristics.

Since it is not possible to actually see the particles we want to study, we instead have to observe the effect that the particles exert on the material with which it interacts in a detector. Creating the right conditions inside a detector allows scientists to see their trails, also called tracks in high energy physics [2]. Many of the particles exist only for a very short time and under extreme conditions, such as when there is very high energy and temperature. To create an environment where these particles can be studied, Pb-Pb and p-p particles are accelerated in opposite directions to speeds close to that of light and collided head-on, creating showers of new particles. The resulting particles will often be heavier than the original particles as they pick up mass from the kinetic energy of the colliding particles.

The energy reached in such collisions at LHC will be in the range from 14 TeV and up to over 1000 TeV depending on the type of collision[7]. The amount of secondary particles created, will for a Pb-Pb collision be several thousands. The collisions happen about 100 meters underground at four experiment-points along a 27 kilometer long

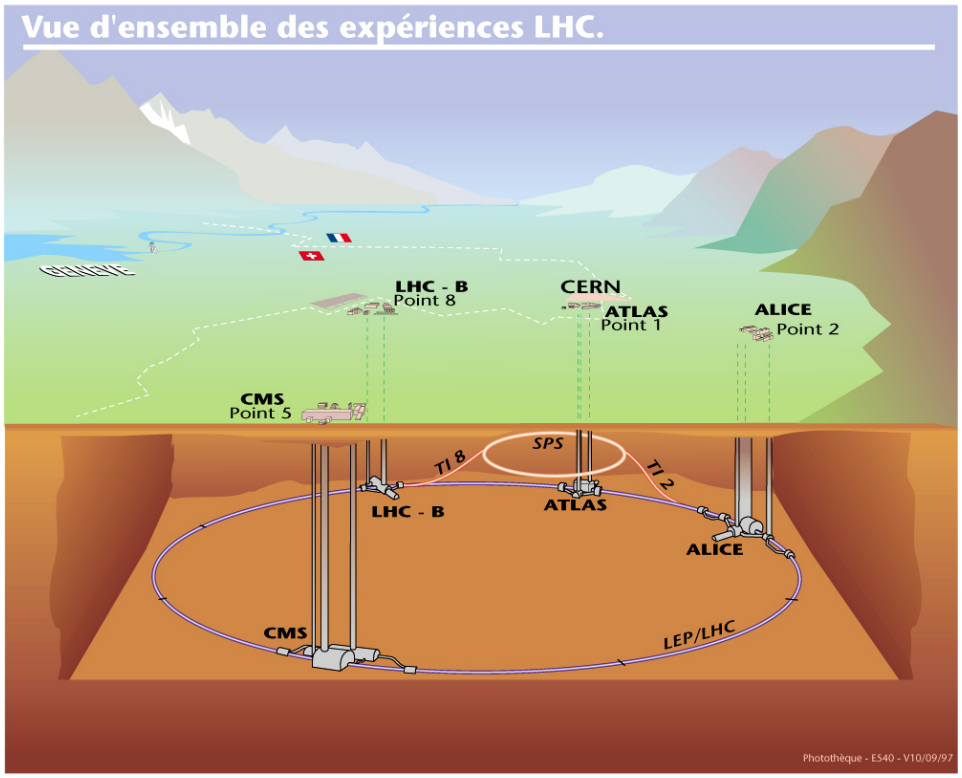

Figure 1.1.: Overall view of LHC experiments[1].

circular tunnel, where all the exeriments are located. An overview of the site is shown in figure 1.1. Apart from ALICE, there is ATLAS, CMS, LHCb, TOTEM, LHCf. All the experiments combined will handle as much information as the entire European telecommunications network does today[7]!

1.1.3. ALICE

ALICE, being one of the experiments, has the highest data rate of all the CERN experiments. From the detectors the data arrives at up to 30 GB/s. This has to be reduced to the available bandwidth of the storage system, which is about 1.2 GB/s. The detectors of ALICE are situated at point 2, one of the four experiment points along the LEP tunnel also shown in figure 1.1.

As with most experiments, ALICE also has several detectors, including: Inner Tracking System (ITS), Time Projection Chamber (TPC), Transition-Radiation Detector (TRD), Time-Of-Flight (TOF), High-Momentum Particle Identification Detector (HMPID), Photon Spectrometer (PHOS) and Forward Muon Spectrometer. All of

them delivering their unique data through fiber cables to counting rooms close to the experiment where computer farms are located. The fiber technology used is commonly known as Detector Data Link (DDL). All DDLs first enter the Data Aquisition (DAQ) counting rooms where they are duplicated and then a copy is sent to HLT. When HLT is done with data processing, the data is sent back to DAQ over DDL. DAQ sees HLT as any other detector that is providing data for event reconstruction (see figure1.2).

While being a general purpose experiment, ALICE is also characterized by being optimized for heavy-ion reactions and is therefore of a very different design to other experiments[2]. The main difference is its primary tracking detector, TPC, which is capable of resolving a very high number of comparatively low momentum tracks.

The resulting particle shower after a collision may travel through several detectors where their effects can be measured. A collision is called an event, and much of the effort involved in creating an experiment is to facilitate reconstructing of these events and recording them in digitized form. The physicist's goal is to count, trace and characterize all the different particles that were produced and fully reconstruct the process and the many interactions[7]. This is to gain a better understanding of how matter was created and what it is ultimately made of[7].

1.1.4. HLT

Because the amount of data produced in all ALICEs detectors is so huge, and many of the events that occur are not important to the experiment, a High-Level Trigger (HLT) has been introduced to help select interesting events (by discarding non-interesting events) and compress data. HLT can provide data aquisition (DAQ) systems with information about which events should be stored, so that data output does not exeed the capacity of storage facilities – the limiting factors being maximum possible taping speed and cost. The goal of HLT is to reduce the stored data rate by at least an order of magnitude while allowing exploitation of the full luminosity[2].

HLT is basically a large computer farm composed of Commodity-Off-The-Shelf (COTS) rack mounted compute nodes. The final installation is planned to consist of about 400 COTS machines, each equipped with dual Central Processing Units (CPUs). The number of CPUs are therefore in the range of 800. There will be about 250 DDLs entering the HLT counting room. Each of these links will be terminated in a Front End Processor (FEP) node. The TPC detector alone will account for 15 GB/s while rate to permanent storage is 1.25 GB/s[2].

1.2. The assignment

For establishing the realtime analysis hierarchy – or more commonly: analysis chain – a Publisher-Subscriber framework has been developed. It provides processing components with an interface to communicate with each other over the local network. The analysis processes will be distributed on the nodes of the HLT cluster. The operation of all these (several hundreds) processes – commanding them to start, stop, connect

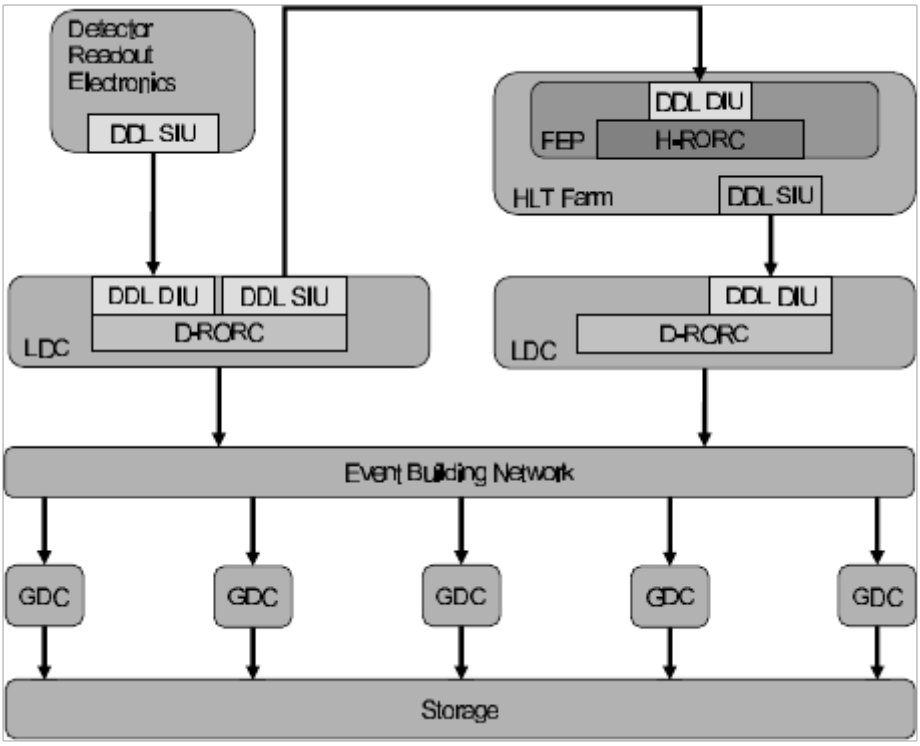

Figure 1.2.: DAQ - HLT dataflow[2].

to each other and so on – will be performed by a task manager, from here on called the TaskManager.

The TaskManager software is made flexible, so that it can take on several roles in a control hierarchy, depending on the configuration file passed on to it upon startup. Python scripts are used to implement a state machine that defines the behaviour and operation of a specific type of TaskManager.

The current control hierarchy consists of a master TaskManager that controls all the slaves. To allow for a more structured hierarchy, a concept of servant and node groups was introduced, but is not yet finished. The idea is that slaves (and node groups) can be grouped in node groups that are controlled by a servant which in turn is controlled by the master. It should also be possible to define more than one servant for a given node group, so that if a servant fails, another servant can take over control of the node group.

To create the proper configurations, a Python program has been developed. It takes a XML (a chain configuration) file as input and produces configuration files (node configurations), one for each participating node. State machines are defined in Python in template files which are mixed into the node configurations.

The purpose of this thesis is to finalize the implementation of the concept of a servant and node groups in these configuration tools. The intention being to avoid bottlenecks by enabling a more flexible and structured control hierarchy.

General improvement of the code, testing and optimization so that the software becomes more reliable and robust is also desireable, but of lower priority.

1.3. Structure of the report

This first chapter has been a short description of the background for the work done in this thesis. It has been an attempt to familiarize the reader with CERN- the organization, the ALICE experiment, HLT, terminology used in high energy physics and otherwise introduce words and expressions that might be new to the reader or deserve to be mentioned explicitly.

The second chapter will be a thorough presentation of HLT, including hardware and software used for its operation.

With the background provided by the two first chapters in place, chapter three will describe the problem in greater detail. A description of how operations are done today and problems with existing methods will be discussed, culminating in a more precise definition of the problem. Finally this chapter will aim to justify the work done in this thesis - why is this problem interesting and what are the prospective advances this work will lead to.

Chapter four introduces problem analysis with a definitive problem definition. Tools, techniques and methodologies will be described and explored to see if they are applicable to the problem domain.

Chapter five moves on to divide the subject matter into smaller, more manageable sub-problems. A solution will be chosen, which should also clearly describe what is contributed and what are already known methods.

Chapter six describes the design, structure and implementation of the system.

An evaluation of the process – with test descriptions and results – and a conclusion that lays out what future work that could be done, are described in the two last chapters, six and seven.

Important words, terms and abbreviations are in general spelled out and explained the first time they appear in the text, but for reference, they can also be found at the glossary in the end.

2. Background

The task HLT performs, is a higly specialized one, not directly connected with the field of informatics. This results in a need for highly application-specific software, implemented in a relatively unknown domain. It therefore becomes important to present the application domain in a thorough way to put everything into context and to make it possible to justify the choices made to reach the goals set in this thesis. The following chapter will therefore provide the background for the rest of the discussion describing both hardware and software of HLT in greater detail.

2.1. Purpose of HLT

The overall mandate of the HLT project is summarized in the following three points[2]:

- Trigger events based on detailed online analysis of its physics observables.
- Select relevant parts of the event or Regions of Interest.
- Compress those parts of an event that are being selected for readout, and reducing the taping rate and associated costs as much as possible without any loss to the contained physics.

The architecture of the HLT cluster has been designed to solve these three points, but in order to find a satisfactory solution the inherent structure of the detectors (in particular the TPC) has to also be taken into consideration. Section 2.2 provide a more in-depth description of detectors and how their data is transported to HLT.

Although the main task of HLT is to reduce detector data into a manageable size, it will also be helpful as a online monitor, for viewing events as they happen and can therefore prove to be a useful tool during, for example, commissioning.

2.2. Detector structure

Experiment data enters the HLT system through FEP nodes that are equipped with HLT - Read-Out Receiver Cards (H-RORCs). FEP receive detector data from custom made DAQ - Read-Out Receiver Card (D-RORC) cards. These cards are physically mounted in DAQ machines and receives data from the experiment and duplicates it for HLT.

For most of the detectors the data processing can in a natural way be done in parallel and in several steps – and therefore lends itself nicely to a hierarchical organization of the nodes. For instance, TPC which will be the detector producing the most data, is

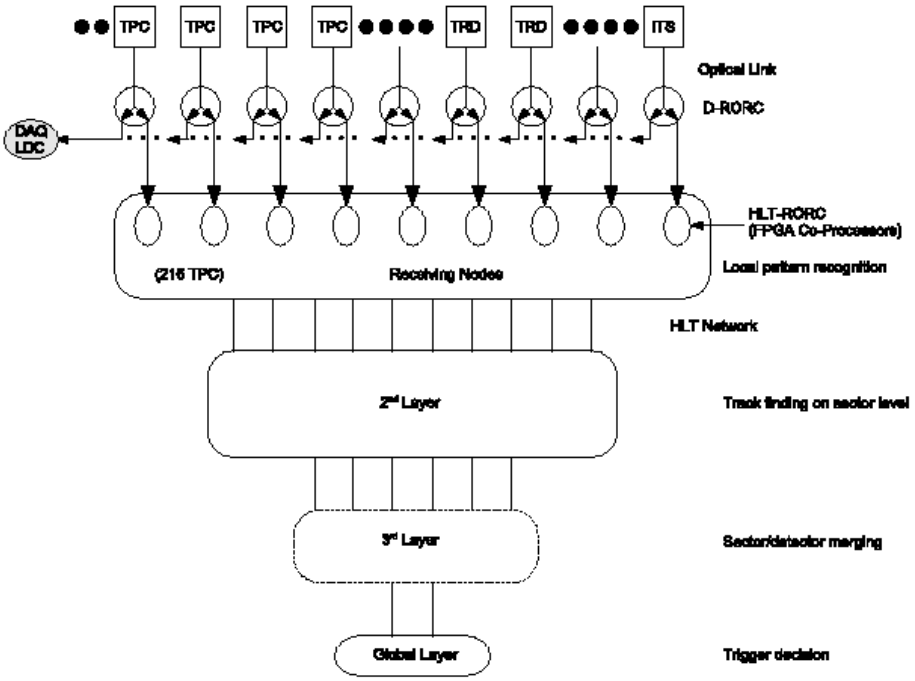

Figure 2.1.: The layers of HLT. Detector data can be seen entering at the top and being split by D-RORC and sent on to HLT[2].

registering hits as particles travels through the detector. Reconstructing these tracks is done in three steps, which are closely related to how the read-out electronics are constructed.

The TPC has a cylindrical shape, with eletronics in each end that read out the data. The two ends are each divided into 18 sectors, which again are divided into six patches. Each of these is connected to one FEP. That makes for 6 * 18 * 2 = 216 FEP just for the TPC.

With this organization in mind, the steps are first to find space points – called clusters — in each patch. These points are in turn used to calculate a track within a given a sector, over all its patches. Finally tracks spanning more than one sector – encompassing the entire detector – are found.

Based on the information derived from this hierarchy of processes, a trigger decision is made and passed on to DAQ. A schematic overview of this process can be seen in figure 2.1.

2.3. Hardware

HLT is a big computer farm consisting mainly of COTS hardware. The nodes are high-performance computers equipped with multiple 64-bit CPU cores and a large amount of memory. The local network uses 100/1000 Mb ethernet and testing has been done to see if InfiniBand[8] can be used.

Although HLT is a software trigger, some hardware has nevertheless also been developed to fill important roles in the HLT system:

2.3.1. H-RORC

In order to be able to deal with the immense processing power required without having to increase the number of nodes beyond acceptable budgets, a specially designed Peripherial Component Interconnect (PCI) card has been created that will be used in FEPs and can perform the initial computations as data arrives. This is possible because the Field-programmable gate array (FPGA) used on this card can be programmed to handle this specific data in a much more efficient way than any general purpose CPUs, even with only a fraction of the clockspeed. The H-RORC cards inserts the processed data into bigphys, a patch for the linux kernel that makes it possible to reserve parts of main memory as a big continous block of raw memory[3]. Because the card is using the PCI standard, it can be inserted in almost any modern computer. This is considerably cheaper than producing a stand alone apparatus for doing similar operations in hardware, because it can be used in current prototypes so that purchase of final nodes can be postponed until very late in the process, taking advantage of Moore's law giving the most processing power for the money.

2.3.2. CHARM

There is one more piece of hardware that has been produced internally in the HLT group and that is the CHARM card. It has been designed with monitoring and maintenance purposes in mind and gives an operator complete remote control over the node it is inserted in. The card exports a screen over the network by pretending to be a graphics card to the node, while also running a small Virtual Network Computing[5] (VNC) server on its built-in CPU over its own network port. Using a VNC viewer remotely, this gives the impression of being in front of the actual node, including viewing boot screens and providing access to Basic Input/Output System (BIOS). The CHARM card also uses the PCI bus to interact with the computer.

2.4. Existing software

Several software packages have been developed for the operation of HLT. Figure 2.2 shows a simple illustration of how theses packages play together on the HLT cluster.

2.4.1. HLT - AliRoot components

Data from the ALICE experiment is processed in two ways: online and off-line. Online means that analysis is performed in real-time such as with HLT in ALICE. Off-line on the other hand offers greater detail, but cannot be performed in real-time.

Components are developed to work with the offline (aliroot) software and then a wrapper is used to make the components seemlessly integrate into the HLT framework. Because of this organization, analysis components will be distributed as part of the ALICE Off-line framework (AliRoot) package.

The analysis components themselves uses ROOT[9], geant3[10] and AliRoot [11] for implementing analysis capabilities. These are all packages mainly developed by CERN.

2.4.2. The Publisher-Subscriber framework

To cater for the flow of data in HLT, a Publisher-Subscriber framework has been developed to be used by all processes to move data between each other independent of running on the same node or not. This design pattern, also called the Observer pattern, is used when cooperating components needs to keep their state synchronized. To achieve this a policy of one way propagation of changes is deployed, so that a publisher notifies any number of subscribers to changes in its state[12]. This fits well with the data driven architecture of HLT, where data is being processed in steps – elements receive input from other preceding elements and produce output data for consumption by succeeding ones[3].

An analysis component typically uses a subscriber object to subscribe to events it is interested in and also has a publisher object where it can publish the processed data to other processes that might be interested in it. In this way, the processes form a chain of data processing analysis components.

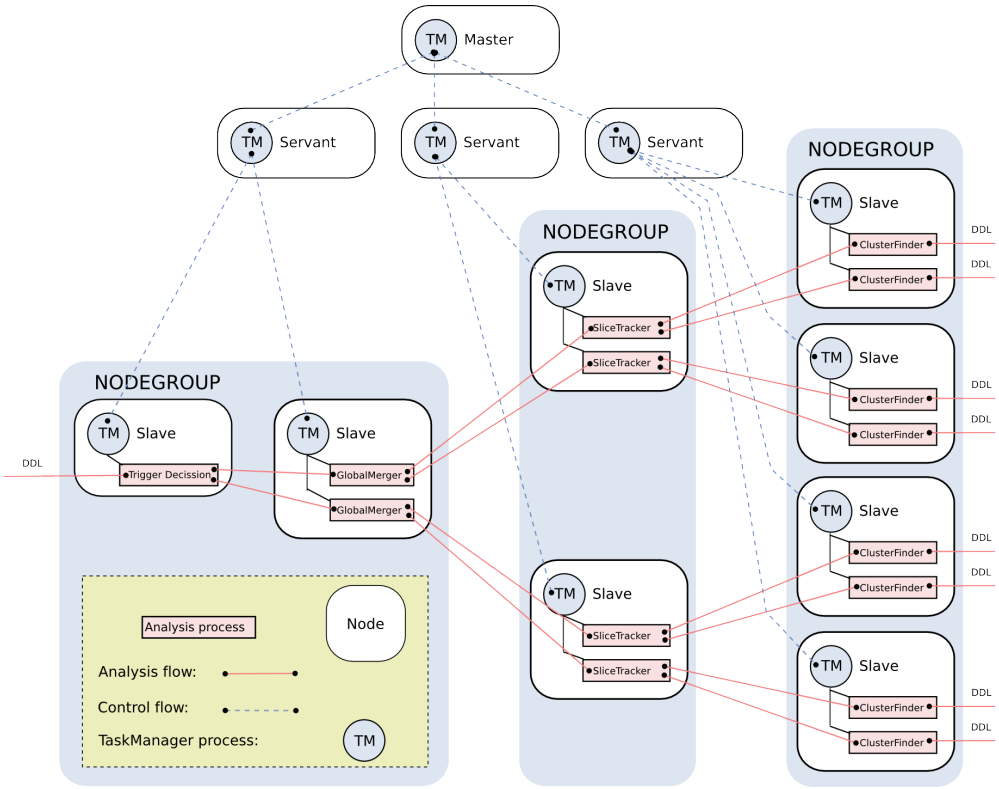

Figure 2.2.: Control and data flow in hlt software, including servant and node group concepts.

Apart from an emphasis on creating a data driven architecture and data driven applications, efforts have been made to ensure that the framework is efficient, flexible and fault tolerant. These being the most desirable qualities of an implementation, have been the main design considerations when creating the Publisher-Subscriber framework[3].

Efficiency Considering that large parts of the framework are simply for moving data around and that the analysis components should be left with the most processing power, it becomes an important objective to optimize the said transportation/routing code for efficiency (use as few clock-cycles as possible, save memory bandwidth at the cost of increased memory consumption).

Flexibility The configuration of the final setup is subject to change as the project moves forward and finally into production. Furthermore, there will be a need to create assorted specific configurations for testing purposes and for adopting to a certain environment i.e. commissioning. With these elements in mind, it is apparent that the framework needs to be flexible so as to be useful in several settings in addition to the foreseen end setup under which it is mainly intended to operate. It is also desirable that the framework can be easily extended to meet future requirements in the case that the experiment is expanded or in any way changed, and for reuse in future experiments.

Fault tolerance A trigger decision in HLT is based on analysis of events reconstructed from several levels of processes communicating either locally or by network. A failure can be of many origins: framework components crashing, lost network connection, failing hardware, complete node crash and so on. As it is impossible to completely safeguard against failure in such a complex system, one will rather have to try to deal with the inevitable in a graceful manner. The goal then becomes to minimize the time from when a lost connection is spotted to the dataflow is reestablished through a new path. To keep the data flowing continuously and without interruption, there needs to be mechanisms in place that can handle failure in both hardware and software autonomously. The Publisher-Subscriber framework has the ability to buffer a certain amount of data, that in case of failing elements can be rerouted to live spare nodes. All connections to the failing node will be replaced by connections to the newly added spare node.

The Publisher-Subscriber framework is composed of generic components and analysis components. Generic components are distributed together with the framework itself and can be divided in three subtypes:

- Data flow components do not modify data as they travel through the system, but manages routing and flow of data. Typical tasks that a flow component would perform might for instance be to merge parts of events into one part, to scatter and gather data among different nodes (load-balancing) or to transport data between nodes over a connecting network[3].

- Worker template components are meant for the users to extend and use in their own implementation. These components can send, receive and produce data (source, sink, analysis). Source and sink can be seen as entry and exit point, respectively, for data in and out of the framework, while analysis is responsible for the actual data processing[3].

- Fault tolerant components are responsible for making the framework resilient to failure, be it failing software components, hardware failure or entire nodes failing. The fault tolerant components are similar to data flow components – essentially being extended versions – performing the same tasks with added functionality[3].

Analysis components are distributed as a separate package, that will eventually be included into AliRoot together with the offline code. These are the components that will do the actual processing of the detector data, from the raw data read out of detector, through a series of components each representing one step in the analysis process, culminating in a fully reconstructed event that is used as a base for the trigger decision[3].

Figure 2.3 shows several components of the framework as it might be used to process TPC data in the HLT cluster. The large rectangles represent nodes and the smaller ones depict component processes. Red components are for analysis, while blue components are generic components. Arrows are used to show the connections between processes.

2.4.3. TaskManager

In an operative (during real run) HLT processing chain there are bound to be thousands of Publisher-Subscriber processes running on the 400-500[4] Symmetric Multiprosessing (SMP) compute nodes. In order to ensure that all these processes operate and interact properly, a control software called TaskManager has been developed[4]. A TaskManager can in turn be controlled by another TaskManager. In this way a hierarchy of control can be established. This hierarchical structure also makes the software scale well with the size of the cluster. The combined goal of the HLT software is to be fault tolerant by avoiding single-point-of-failure and central bottlenecks.

Well known technology is an integral part of the software, XML being used for configuring the TaskManager instances and an embedded Python interpreter being used to implement a state machine for run control. Interface libraries are used to access the controlled components[4]. An overview of the TaskManager internals is shown in figure 2.4.

There are three subsystems close to the core that provides most of the functionality of the TaskManager. This is the Configuration Engine, the Program State and Action Engine and the Program Interface Engine. The configuration engine holds the configuration read in from the XML file and makes it available to the other modules. The embedded Python interpreter is used by Program State and Action State Engine to execute the Python code contained in the configuration files. The Program Interface Engine performs the actual communication with the programs this TaskManager

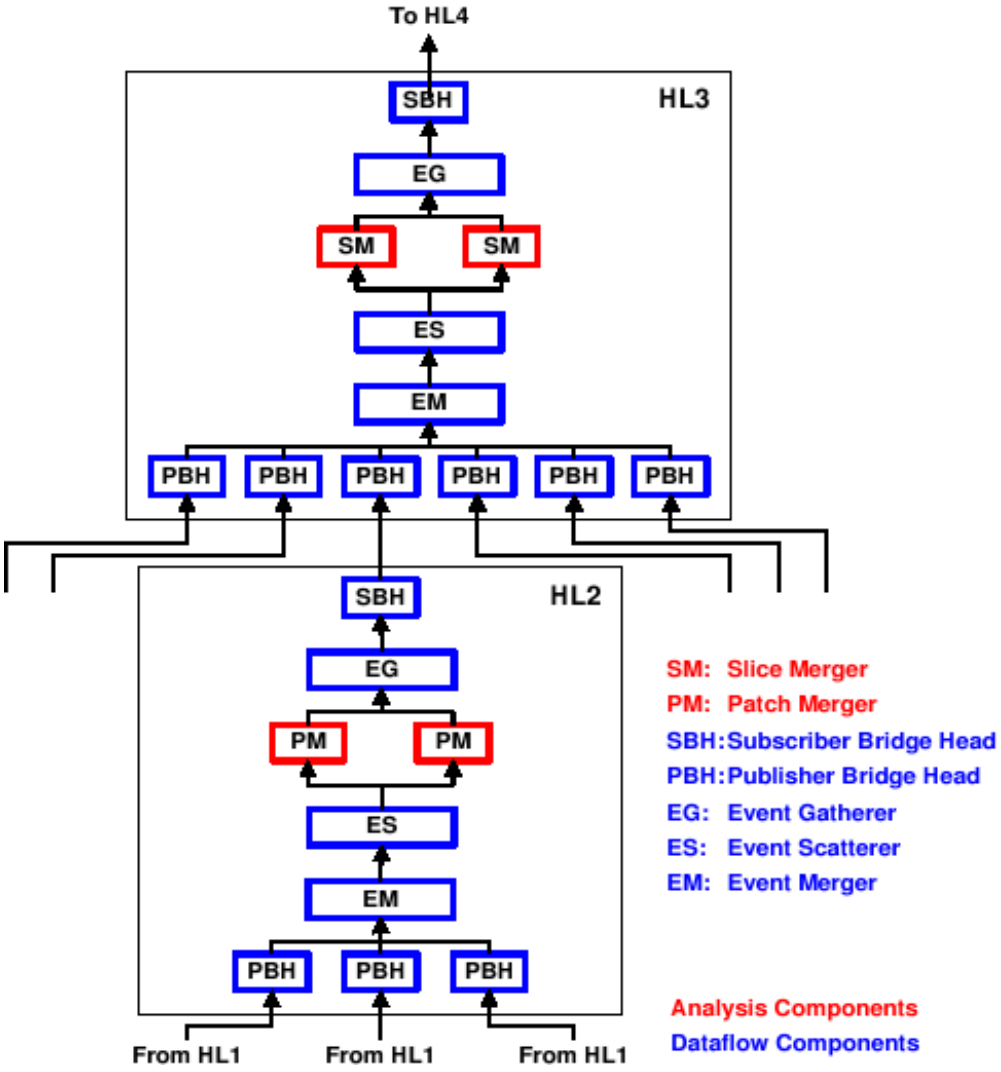

Figure 2.3.: Example of components in Publisher-Subscriber framework[3].

instance is in charge of. This is done through external interface libraries, which is supplied to the program at start up[4].

Configuration Engine A configuration holds several pieces of information vital to the operation of the TaskManager. Most importantly, the commands for the programs to be started and Python code for the state machine. Some of the configuration entries are global, such as the interface library to be used and Python actions valid for state change in any program. If the configuration is hierarchical, there will also be slave (TaskManager) entries defined globally.

The program entries for each of the processes to be controlled, holds commands to execute the program including command-line parameters – either a TaskManager or an analysis process – and the address of the process. Also Python actions defined for these programs events are defined here. (All these programs are for defined process only and not valid for any other processes). Several program events are present in the system, for which Python code can be assigned: status change, program termination, changes in program configurationentry.

Program State and Action Engine The state machine implemented as Python actions defined in a configuration file and an embedded Python interpreter, makes for a very powerful and flexible solution. Events are defined in the system that will cause the Python actions to be executed by the interpreter. The relevant functions in the TaskManager are exported and made available to the Python interpreter and can be called by Python code like any other Python function. Python code can therefore easily interact with the TaskManager system and controlled programs. Example of some of the operations made available to Python including query state and sending commands.

Program Interface Engine The flexibility of the Program Interface Engine is ensured by making use of functionality made available by shared libraries dynamically loaded during start up. The libraries implement an interface of standard operations for each of the program types the TaskManager has to deal with. This way many different program types can be handled in a coherent way by the TaskManager.

When loaded, the functions of the shared libraries are available as normal from C/C++. Additionally, the interface library functions for querying program state and status data as well as sending commands are also available to Python.

For the operation of slave TaskManager, by master TaskManager, there is one extra component active on the slave. This component is responsible for dispatching commands received from the master to appropriate slave TaskManager subsystems, which is either a slave TaskManager or an analysis component. It also provides status information to the master.

The master, on the other hand, has no extra objects in use, but rather uses the basic mechanism of communication with controlled processes via an interface library. A special interface library is provided that supplies the required support for master-slave TaskManager communication. TaskManagers are specified as any other process to be

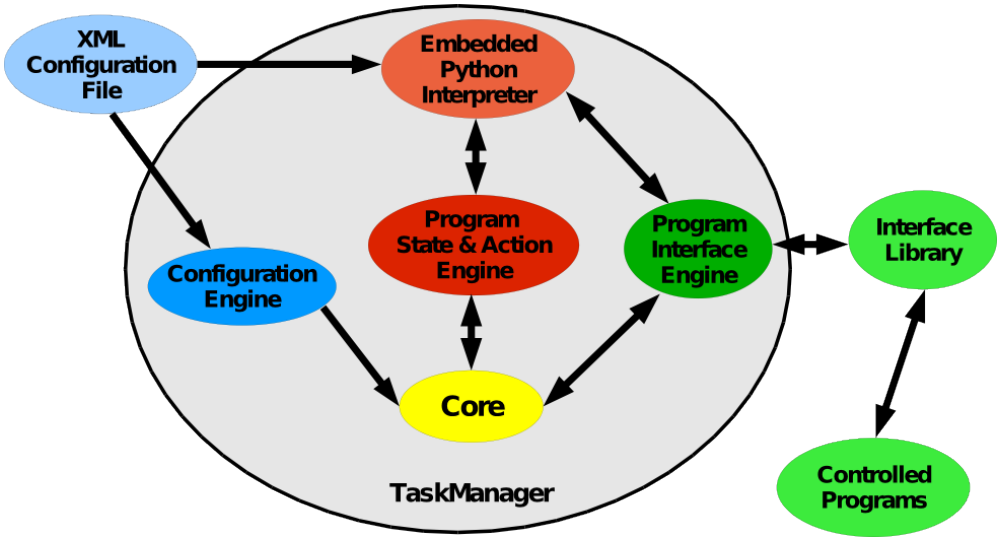

Figure 2.4.: Internals of the TaskManager[4].

executed under the master TaskManager's control. By reusing the default mechanism the required configuration item support is reduced to a minimum on the master[4].

Most of the design considerations for the Publisher-Subscriber framework are also valid for the TaskManager as they will work together to fulfill the software requirements for HLT. The TaskManager needs to be flexible and fault-tolerant for the very same reasons that the Publisher-Subscriber framework needs to. It is also designed to increase manageability as the several thousand processes are not manageable in a sensible way by a single supervisor instance[4]. The hierarchical organization of control made possible by TaskManager also enables partitioning and eases configuration of the system[4]. Lastly, the system should not make any significant impact on the performance of the analysis processes.

The TaskManager software is used to control all the involved tasks in a configuration. Every node has an instance of the TaskManager program running which controls all the components of the Publisher-Subscriber framework running on that particular node. The TaskManager can be configured to operate either as a slave or as a master TaskManager, depending on the configuration passed on to it. The master TaskManager is the single point of control for the entire chain and this is where clients for control can connect and interact. Commands given to the master TaskManager are translated into suitable commands for the slaves and passed on to them. Likewise, when the slave TaskManagers receive a command it is translated into commands for its child processes and passed on again. This way, commands are propagated all the way down the hierarchy from the client program interacting with the master TaskManager, to the analysis components. Figure 2.5 shows data and control flow in a small setup of HLT software.

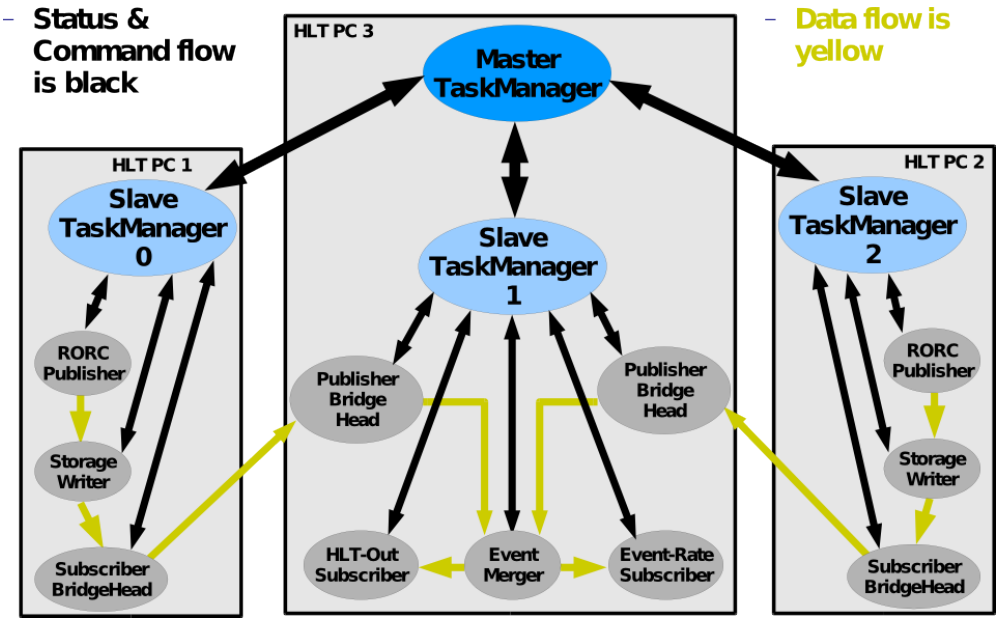

Figure 2.5.: TaskManager control and data flow[4]. A simple master, slave control hierarchy is depicted. Large rectangles represents nodes. Ellipses are processes.

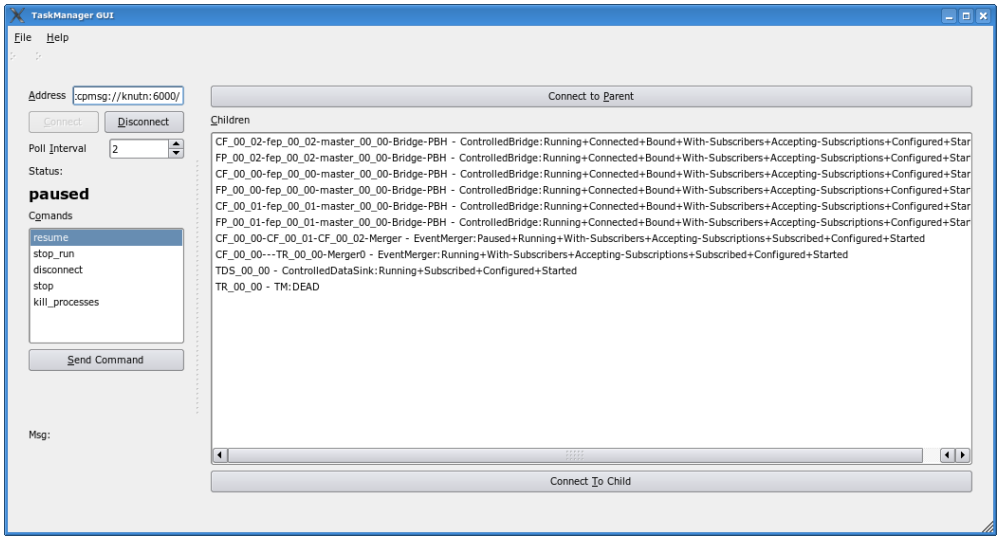

Figure 2.6.: Screenshot of TaskManager control GUI.

Typically there is only one TaskManager on every node, but the node with the master TaskManager can also have a slave TaskManager with its accompanying Publisher-Subscriber processes running along with the master. A TaskManager is started by bash scripts created by configuration scripts and configuration files created by the very same configuration scripts are passed as arguments. Bash scripts are also created for testing and stopping a chain.

2.4.4. TMGUI.py - Task manager GUI

For easy, intuitive control of the chain for end users, a simple Graphical User Interface (GUI) has been created. It is a Python program using PyQt[13] Python bindings for Qt[14] to draw the GUI elements. A screenshot of a running instance is shown in figure 2.6. To the left, the status of this particular process is shown and a list of commands that can be executed for the children of this process. In the larger view to the right all child processes are shown with the current accumulated status at the end of the line. There are buttons to connect to parent or child, meaning the GUI will change its view to show the respective information. The initial view is typically from the master TaskManager, showing all slave TaskManagers as child processes. In the screenshot a slave TaskManager has been highlighted running on the master node, the same node running the master TaskManager.

The GUI program can parse the master configuration file to discover the address and port of the master TaskManager which it should connect to. The connect/disconnect buttons handles the connection with the master TaskManager. Lastly, there is a setting for poll interval, which tells the GUI how often it should query information from the

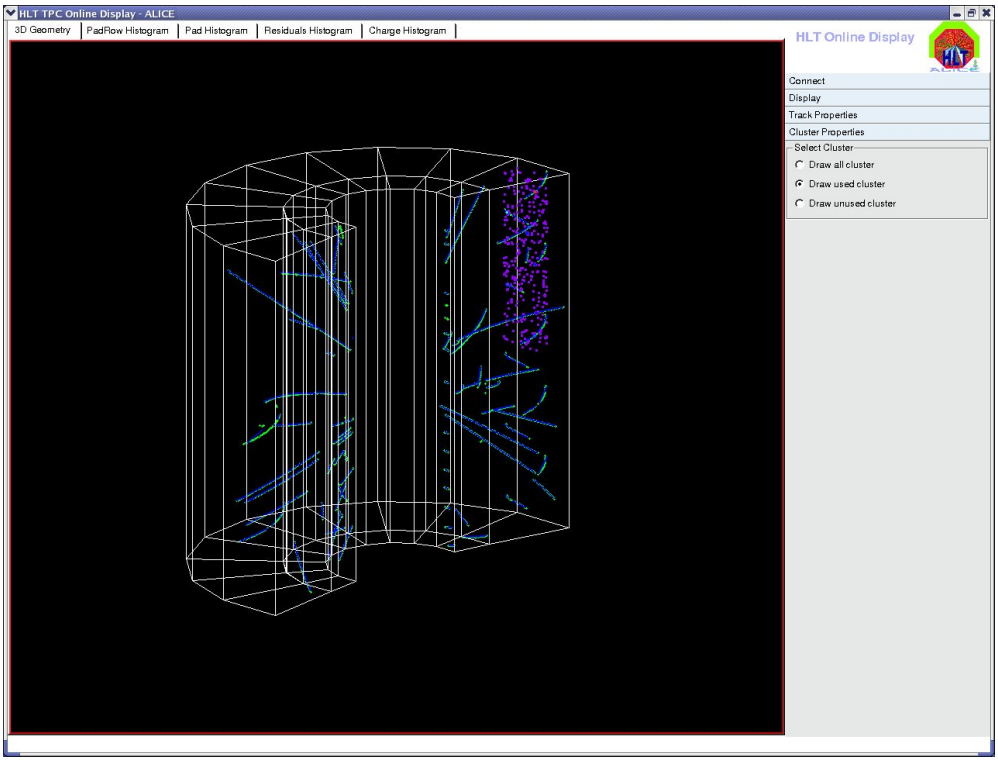

Figure 2.7.: Screenshot of TPC Online display[1].

TaskManager.

There also exist simple command line client programs, that are useful for debugging for instance. All of these client programs connect to a master TaskManager and use a library with Python bindings to send commands and retrieve information.

2.4.5. HLT TPC Online display

The HLT TPC Online display is a GUI that visualizes single events in the TPC detector as they happen. A Publisher-Subscriber component (TPCDumpSubscriber) can be prepared in the configuration to dump data to a socket to which the Online display can connect and visualize the received data. A screenshot is shown in figure 2.7.

2.4.6. Configuration tools

The last piece of software used to operate the HLT is the configuration tools for creating the XML files used for controlling the TaskManagers. This is done by Make-TaskManagerConfig.py, a Python program that takes a single chain configuration file

and generates the beforementioned node configurations files and run-control scripts. The configuration tools will be presented in greater detail in chapter 3.

2.5. Fault tolerance in HLT software

For successful operation, HLT needs to be highly fault tolerant. If a node fails, it must be quickly replaced so that data is not lost and the system can continue to operate. The FEP nodes have an elasticity buffer that can accumulate events. Given a memory size of i.e. 2 GB for each DDL, this buffer can hold 20 seconds worth of data taking[2].

A solution for increaseing fault tolerance have been tested and might provide indications of how fault tolerance will be implemented in the final setup. With the software in this test, the processes of a failing node will immediately be distributed to a set of predefined nodes, while a spare node is prepared with a suitable configuration. When ready, the new node will assume responsibility over the temporarily moved processes, re-establishing the original structure[3]. To be able to do this, the components used on the nodes will have to be prepared for fault tolerant operation as described in section 2.4.2.

2.6. Known methods

The configuration tools have been born out of the need for the developer of TaskManager to automate the process of creating the configurations that describe a certain setup. The very specific nature of the TaskManager, makes it hard to find examples of how similar tasks have been solved before. It is about configuration of parallel distributed software. Important aspects of the implementation are flexibility, adaptability and extensibility.

The configuration of HLT processing chain needs to be flexible and easy to change. Many different setups will be tested so that an optimal organization can be found. It also must be easy to adapt the setup to new detectors as they are added to the setup and to extend the tools with new features as they are needed.

Studying source code and actual usage of the software will propably be the most useful guides to a satisfactory implementation.

There have been certain attempts at developing tools that can aid the creation of the initial chain configuration file, since, this is still a laborious manual process. A recent thesis tries to address this problem by developing a web-based interface which can generate a configuration based on choices made by the user[15].

3. Configuration tools

This chapter discusses the configuration tools in greater detail and present the new requirements for the software. In addition, the first two sections provide a short background for creating the configuration tools in the first place and the last section explains why this task is interesting.

3.1. Control hierarchy in HLT software

The flexibility that the design of the TaskManager allows, makes it possible to consider a broad range of solutions for implementing a control structure in HLT. Managing of new types of processes can be introduced by implementing new libraries (analysis processes vs. other TaskManager processes) that can be dynamically loaded by a TaskManager during startup. To implement new types of nodes (as a state machine), it should be enough to create the appropriate Python template files and possibly adopt the configuration tools.

Initially a master and a slave state machine were implemented (and libraries for controlling TaskManager processes and analysis components). With these two types of nodes, a simple structure of a single master controlling all slave nodes, can be created. To enable a more structured hierarchy – that is also less prone to cause bottlenecks – two concepts have been foreseen: node group and servant. These are already partially implemented, but some work is left to make these concepts complete.

A servant is a particular type of node that will be in charge of a node group, for which it will start, stop and configure its slave nodes as it is asked to. If a servant node fails, other servant nodes will be lined up to take over control of its node group.

A node group is a collection of nodes that naturally belong together and are controlled in a similar fashion, so that a single servant instance can assume responsibility over this node group with a suitable command set.

There are two types of communication flow or hierarchies in HLT. The Publisher-Subscriber framwork is concerned with data flow in the form of detector data, while the TaskManager is concerned with control flow, as it manages the operation of HLT software.

3.2. Motivation

For simple setups, creating configurations by hand is a cumbersome, but still manageable task. The duties of scripts and configuration files evolve along with the development of the software itself. For testing simple concepts, few or even a single

node might suffice, so there is no real need for an automated assistance in the form of specialized configuration generating software. As the project grows, the need to create more elaborate tests, involving a larger set of elements, arises naturally from the need to stress the mechanisms involved with overall operation, connectivity and for complexity's own sake.

Considering the above-mentioned growing complexity and the design goal of being flexible, a natural development of the initial handwritten configuration files is to evolve into software suitable for configuration creation. In a somewhat natural way, this has taken the form of a template based process, where an object oriented Python program takes a simple configuration file as input and based on the definitions therein generates all other files by mixing the templates into the output.

The XML file describes, among other things, what commands each TaskManager should run for each of its child processes and what kind of shared memory should be used for its child's communication. Further, it dictates which node a given task should run on, connects tasks to each other and sets up the entire hierarchy of control amongst nodes and processes (an example can be seen in appendix B).

This initial configuration is capable of holding a description of an entire configuration, but is not a description of the structure itself in such a way that the amount of configuration information that needs to be written is minimized. Essentially, mapping is missing. That is, the ability to map processes to nodes automatically, based on some criteria – preferably specified inside the configuration file – is not in place. The configuration file is the mapping, not a description how mapping should be done.

Therefore, although a very large part of the work is automated (run-control scripts and node specific configurations are automatically generated), there still is a lot of manual work involved in creating larger configurations, which can span i.e. thousands of lines for a simple configuration of one side of the TPC detector.

3.3. Implementation

While the TaskManager and HLT software in general has been designed with fault tolerance in mind, the configuration tools do not yet have the functionality implemented to configure the TaskManager for a fault tolerant setup.

MakeTaskManagerConfig, being the application that the user executes, is mostly dealing with parsing command line arguments and ensuring proper execution by informing the user of errors. A configuration reader is used to parse the configuration file and build the SimpleChainConfig object from SimpleChainConfig1.py. The returned SimpleChainConfig object is used by a ChainConfigMapper object to map platform and network specific information to the nodes in the configuration object. In the end, the TaskManagerOutputter and its derived objects/classes are generating the actual configuration files and run-control scripts that are used to configure and operate the TaskManager.

Refactoring, structure of configuration scripts Apart from the templates, most of the code is split between two files, MakeTaskManagerConfig.py and SimpleChainCon-

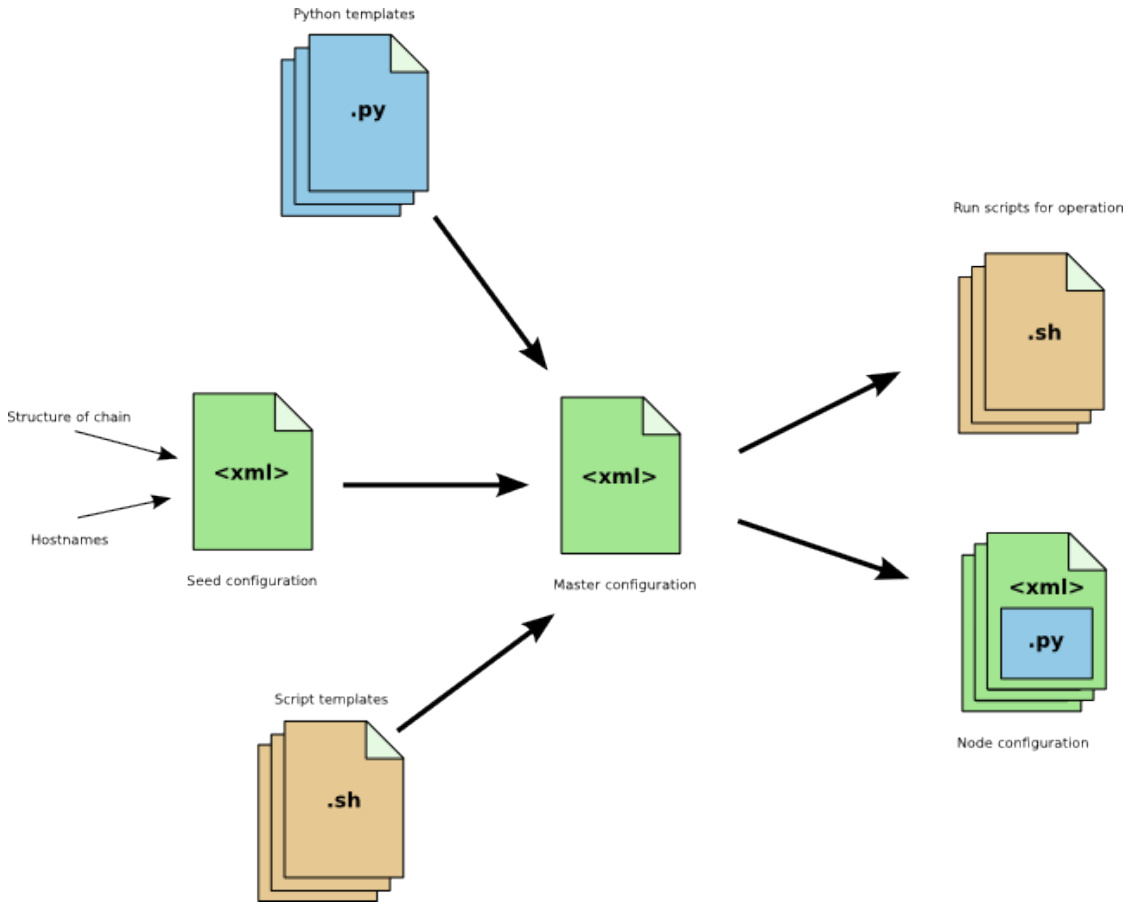

Figure 3.1.: Process of creating configuration files.

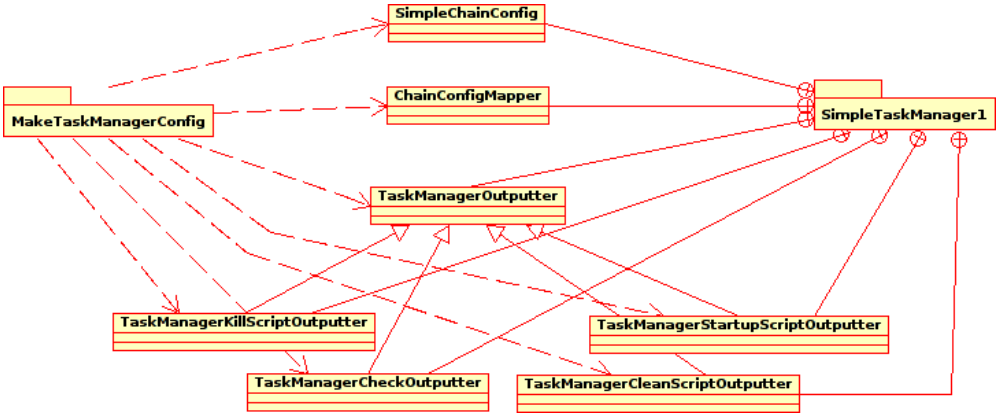

Figure 3.2.: Class diagram of configure script.

fig1.py. Since Python is a very flexible language with respect to how code can be organized in files, an implementor can choose to have all code in its entirety in a single file. Code can just as easily be divided into files, one for each class. For the configuration scripts, all classes resides in SimpleChainConfig1.py. It might be beneficial to split SimpleChainConfig1.py into several files for practical reason, to have the code more easily accessible in a text editor. Benefits that naturally follows a refactoring are for instance increased readability, better reusability and extendability, improved general structure and reduced dependability[16]. Despite the relatively small size of the source code in this case, agile practices suggests that refactoring should always be done when code can be improved.

3.4. New requirements

The complete set of functionality to be implemented for the system is presented in greater detail as user stories in chapter 5.1. Although, also user stories are sparse on implementation detail as defined by Extreme Programming (XP) practices. Tasks from the customer are first here presented as functional and non-functional requirements, and later refined and transformed into user stories. Requirements and tasks that emerge with the progress of this thesis, will be specified directly as user stories.

3.4.1. Functional requirements

Functional requirements describe the inner workings of a system. How its internal machinery is intended to operate and how it is supposed to respond to interaction with external entities such as users and other systems[17]. For example, in an invoice system, a functional requirement would specify that customer information should be stored in a customer database. The functional requirements do not however specify the details of the database or how it should be implemented.

Finalize implementation of servant/node groups The servant and node group concepts was introduced to enable a more structured control hierarchy in the TaskManager. The increased flexibility added by these two elements will for example make it easier to create hierarchies that are less susceptible to bottlenecks.

It should also be possible to make several servants supervise a node group. So that if one servant fails, another can take over. Support for this is already present in the framework. What has to be added is functionality for creating the additional servant nodes and the proper connections between node groups, servants and master nodes with the configuration tools.

The servant and node group concepts should be completed by modifying the respective template files and the configuration tool itself. The implementation should also be expanded so as to enable more than one servant to supervise a given node group.

Avoid recompile of Python bytecode The first time a Python program is executed, a byte code representation is stored in a file that can be directly loaded into memory next time the program is started, thus reducing load time. This happens transparent to the user as the Python interpreter checks to see if the Python source file has remained unchanged since last run, and if so, loads the byte code instead of the source file. During configuration of the TaskManagers loading of Python code happens repeatedly, but the current design does not allow for taking advantage of reuseable pre-compiled byte code.

If a way is found to make use of this capability of Python, the speed and general responsiveness of the system would be improved.

Separate logging Seperate logging output generated by configuration tools from the TaskManager output.

3.4.2. Non-functional requirements

While functional requirements describe what should be performed and how, non-functional requirements are criteria to define how good a certain quality needs to be[17]. Robustness, speed, memory usage and cost are all examples of non-functional requirements. Generally, constraints imposed on the system by involved factors, such as programming language and development platform, are also considered non-functional requirements.

Efficiency, flexibility and fault tolerance The non-functional requirements defined for HLT software earlier in chapter 2.4.2 are also valid for the configuration tools. The efforts put into the other HLT software should not be made futile by not conforming to the same requirements that of i.e. TaskManager and Publisher-Subscriber framework. In line with the efficiency requirement, one of the tasks from the customer is to study execution paths with a performance profiler to see if it is possible to increase efficiency in the configuration tools:

- Check which steps take time with the original configuration tools and see if it is possible to optimize efficiency.

Operating System Much of the software systems developed for the experiments at CERN are using some sort of linux distribution. Although CERN has its own linux distribution (Scientific Linux CERN[18] (SLC)), HLT will be using the ubuntu[19] distribution on the cluster. Ubuntu will therefore be the target platform for application development in this thesis.

Programming language The configuration tools are written in Python and uses also Python for its templates. Any extension to these scripts must therefore also be written in Python. The two main programming languages within the project are C++ and Python. For the overall project, using few programming languages helps to keep complexity low and increase consistency.

3.5. Why interesting?

The LHC experiment is on one level providing bits of answers to the big questions humans have asked throughout history. Questions such as: Where do we come from? How did everything start? And what is everything ultimately made of? The results gathered from LHC might help guide scientists to answer questions like these. It is a privilege to have the opportunity, even to play a small part in this groundbreaking endeavour, and the fact that the LHC is nearing completion makes it even more exciting. The complex nature of the HLT software makes it a challenging and interesting domain for working on a thesis. A deep understanding of the system is required to achieve favorable results. The primary goal of providing configurations for improved reliability is a worthwhile undertaking as reliability is highly critical for the success of HLT. As the project goes forward, it will be more and more important to ensure that the system will work reliably during long runs. Worthwhile is also the secondary, less specific goal of general code base improvement and optimization, which in addition also is a very sensible consideration for the long term.

Lastly, the tools, technologies and methods used for the development of HLT software are highly relevant in todays IT environment. Agile practices are encouraged through use of agile languages such as Python. Also, the extensive use of Linux within the experiment, caters for a familiarity with Unix-like systems that will be a valuable asset as Linux is on the verge of larger mindshare and adoption.

4. Problem analysis

This chapter will begin with a precise and final formulation of the work assigned for this thesis. The methods and technology that might be useful for resolving the problems will be discussed next and the third section consist of considerations made with regards to development of the software in this thesis – including implementation guidelines, automated build tools and development environment.

Whenever the word "customer" is used in this text, it refers to the person or group who assigned the task for this thesis. As one of the principal stakeholders, Institutt for Fysikk og Teknologi (together with Kirchhoff-Institut Für Physik in Heidelberg) acts as the customer on behalf of the ALICE experiment in this case.

4.1. Problem definition - precisions

From a software engineering point of view, this thesis has two categories of problem it attempts to resolve. The first category covers implementing features in order to alter the behaviour of the configuration tools (add desired features). The other category looks at improving structure and other properties of the software without changing its behaviour:

Feature adding goals:

- **Finalize servant/node group implementation:** The servant/node group concept is intended to enable a more structured control hierarchy of TaskManager. Initially there are two tasks that need to be completed before the servant/nodegroup implementation is complete:

  1. Servant should start all slaves (can be both servant and slave TaskManagers) belonging to its node group upon startup.
  2. It should be possible to specify more than one servant as "master" of a node group. A master of nodegroup translates into a servant by the configuration tools. The configuration tools should then output configurations that has one servant that runs per default, but also one or more additional servant that are latent nodes that can be activated if the master TaskManager senses that the primary servant is inactive i.e. that it has in some way failed.

- **Explore alternative implementation:** As the configuration tools developed gradually from hand written files, it might, with the knowledge now available after development and use of the configuration tools, be beneficial to consider

a different structure as a basis for an alternative implementation. Writing an initial prototype that explores a different approach might provide an indication of whether a different structure might be worth considering.

Properties improvement goals:

- **Refactoring:** There are several symptoms of code smells in the current configuration tools: Long functions, long parameter lists passed to functions, lots of local variables, little reuse (not making full use of built-in Python libraries or libraries with Python bindings), code duplication, hard to make changes and large parts of code collected in few files. Refactoring the source would create a more usable tool for the future.

- **Optimization** Investigate if there is room for improvement in efficiency. Using a profiling tool it should be possible to see which parts of the code use the most execution time. This might provide indications for where optimizations could be made.

Chapter 5 will divide the goals mentioned above into smaller problems and present them in greater detail as user stories.

4.2. Methods and technology

4.2.1. Practices and methodologies

As software engineering outgrew its infancy, methods emerged that sought to put the process into system and make its progress measurable. Traditionally, this has been done by creating processes to handle the process itself. These processes are generally thought of as reminiscent of a waterfall, where tasks are finished one by one, and a task is not started unless the previous task has been completed, thus they are also called waterfall processes[17].

Although not being entirely new ideas[20], iterative and agile methods have recently gained popularity and received much publicity. Iterative methods repeat most of the single tasks several times throughout a project. The main purpose is to avoid costly changes late in the process by continuously and repeatedly refining the outcome at each step. A better control of change, without the frustration the heavier processes seems to be burdened with is what proponents hope to achieve[16].

The most recent trend within software methodologies is the Post-Agilism movement that seeks to avoid being constrained by the Agile Dogma[5]. It lends from a larger set of methodologies and recognizes more general ideas such as the the importance of self organizing teams vs. the hindrance rigid control from above can cause. Also described as Nonlinear and Linear Management [5], where linear means order is achieved by manipulating the different aspects of a business (process for instance), while nonlinear relies on order to emerge by itself, when teams are self-organized and are allowed to evolve and adopt to suit its tasks.

For the work with this thesis, the most relevant methodologies are considered to be Unified Process (UP) and XP. Both of which are considered to be iterative or evolutionary models as opposed to the two other classes of process models: waterfall and component-based engineering[17]. It is expected that the model of working with this thesis will embody elements from both XP and UP.

Extreme Programming One of the premises for using XP is that unless all practices are being applied, it can not be considered proper XP with all its benefits. That does not, however, mean that using only some of the practices cannot be useful. Several of the XP practices should be possible to successfully apply to the work with this thesis.

XP has a set of clearly defined practices that should all be followed to achieve the best results[16].

1. **Customer Team member:** The customer is the one who defines and prioritizes the features of the software. He or she should therefore work very closely with the team, preferably be a team member.

2. **User stories:** Instead of requirements in the usual sense of the word, XP utilize user stories to describe aspects and functionality of the system. The main purpose of a user story is not to describe a feature in great detail, but rather to describe it sufficiently to be estimated. One user story typically describes one feature and when the customer and developer has agreed on the user story, the developer estimates its cost.

3. **Short Cycles:** XP uses short cycles and delivers working software at the end of each iteration. Usually an iteration is two weeks. For each delivery the system is demonstrated to stakeholders, who give feedback to the developer team. There are two types of plans involved in a XP project. The Iteration Plan spans an iteration and is a collection of user stories selected by the customer according to a budget established by the developers. The developer sets the budget for an iteration by measuring how much got done in the previous iteration[17]. The other kind of plan, is a release plan which spans approximately six iterations. Usually three months work. The content is similar to the Iteration plan, but at a larger scale, and in contrast to the Iteration plan, the Release Plan can be changed at any time by the customer.

4. **Acceptance Tests:** Acceptance tests are used to capture the details of user stories. They are written before or at the same time as the implementation of a user story. The acceptance tests are usully written in a scripting language, so that it is possible to run them automatically. Once an acceptance test is passed, it is added to a list of test that are never allowed to fail again. These tests are run everytime the system is built. The outcome should be that once a feature is implemented, it should never be allowed to be broken.

5. **Pair Programming:** All production code is produced by developers working in pairs at the same computer. One person “drives” the session by writing the

actual code, while the other is watching for errors and improvements along the way. There is a continuous discussion about the code being produced. The pair exchanges roles frequently for instance if the driver gets tired or stuck, the keyboard is passed to the other developer. Pairs change at least once per day, so that everyone works in at least two different pairs. During one iteration, all the developers should have worked with each other and on all aspects of the system. This increases the knowledge about the system throughout the team. The end result is reduced amount of defects in the software, while efficiency stays the same.

6. **Test-Driven Development:** First, unit tests are written for a piece of functionality, then production code is written to make the unit test pass. The developer iterates between writing test and code within minutes. A large body of tests are being created alongside production code, which facilitates refactoring. Test-driven development also encourages decoupling modules.

7. **Collective Ownership:** All pairs are free to check out any module of the system and work on it. Developers are not individually responsible for or have any authority over any piece of code. This also helps to spread the knowledge of the system.

8. **Continuous integration:** XP uses nonblocking source control, so that anyone can check-in code at any time. The first one to check-in wins, the rest have to merge. For every check-in the developers have to check if there are merges waiting, possibly do the merging, then run all tests and if they successfully complete, first then the code can be checked-in.

9. **Sustainable Pace:** In XP, development is seen more as a marathon than a sprint. Overtime is not allowed, except for the last week in a release. By only using the allowed budgets and adjusting the budgets for every iteration, the pace is kept at a constant rate.

10. **Open Workspace:** The entire team works in an open room, with workstations prepared for pair programming spread throughout the room. Walls are used for diagrams, illustrations, etc. Everyone is within range for discussion or questions.

11. **The Planning Game:** The planning game is with its simple rules what makes XP work. This is how business people have means to keep track of the development and developers get to communicate the complexity of the system. The customer will have a budget based on how much was achieved in the previous iteration. The developer assigns cost to user stories and the customer chooses user stories according to his available budget for the iteration.

12. **Simple Design:** The mantra for simple design is to not add features that do not add value to the system. A design should be kept as simple and expressive as possible. Only user stories for the current iteration is taken into account when designing. There are three guidelines to follow: *consider the simplest thing that*

could possibly work, *you are not going to need it* and *once and only once*. These guidelines advice the developer to keep things simple, avoid unecessary features and avoid code duplication.

13. **Refactoring:** To avoid code rot as features are added and bugs are squashed, frequent refactoring should be done. To refactore is to improve the structure of a program while not changing the behaviour. Refactoring is done incrementally in small steps and is a continuous practice to use throughout a project. Refactoring keeps the code clean, simple and expressive.

14. **Metaphor:** The metaphor is the big picture; the vision of the system that makes the location and shape of all the individual modules obvious. The metaphor is often manifested in the form of a set of names describing the system. It provides a vocabulary for the elements of a system and their relationships.

Some XP practices are impossible to use in the context of a master thesis. For instance, since a thesis usually is limited to a single person, practices that involve more than one person – such as pair programming – are hard to exercise. Others have limited benefits; working in an open office environment is not neccessarily going to be beneficial when working with a thesis.

Some practices are easily applicable to most domains of software development and can serve as guidelines independent of the chosen development methodology. Attention to code quality and striving to continually improve the source code will most certainly have a positive effect on the overall software quality.

Practices that could easily, and probably with a favorable outcome, be used in this thesis are short cycles with release of working code, test driven development, user stories, planning game and acceptance tests.

Other prominent agile alternatives include scrum that are characterized by a scrum master that shields the team from outside distractions[21]. Otherwise it uses a different vocabulary than XP for many common concepts and has a larger set of different meetings to keep track of progress, while having fewer practices than XP.

Unified Process Unified process is an iterative and incremental development process. It has phases like a waterfall model (inception, elaboration, construction, transition), but are divided into a series of timeboxed iterations, each iterating – to various degree – over all processes of software development. That means that requirements, design, implementation, test and project management activities are all part of the work done in a given timebox and the activities are not divided in sequential steps that have to be completed before entering the next phase. Unified Process has a strong focus on architecture and makes use of several perspectives to build a proper understanding of the system. Risk is as with agile methods in general, dealt with early on in the project. Use cases are used to describe functional requirements.

4.2.2. Programming languages

The configuration tools are written in Python, therefore an extension will inevitably have to use some Python. But it is fully possible to combine Python and other languages, even compiled ones in many ways. The TaskManager for instance is written in C++, but contains a Python interpreter that is used to interpret Python code loaded from the configuration files. This Python code implements a state machine for the given TaskManager and uses a provided Python interface (TMLib.so) to perform actions on the TaskManager.

The properties of the different programming languages makes them suitable for different tasks. For instance, a program written in C++ will most certainly run faster than the same program written in Python. However, the time it takes to write a Python program is likely to be much shorter than the equivalent C++ implementation. The typical Python program is also shorter, clearer and therefore more maintainable. So depending on which properties a piece of software should have, the developer can balance the amount of the different languages accordingly. Usually this is a speed vs. productivity and maintainability trade-off.

Programming languages can in general be divided in two broad classes with respect to how their code is executed. There are interpreted languages that need an external program to interpret its intentions on the fly while executing and there are compiled languages that run directly as machine code without any further assistance than the Operating System (OS). Some languages, such as Java[22], can be seen as a hybrid; it is compiled into byte-code, but still needs a runtime environment to run.

There are other developer languages too that are not programming languages per se, but that are quite useful tools for a programmer. Either as a communication tool of concepts and models (Unified Modeling Language[6] (UML)) or as a language describing data in a well specified way (XML). The rest of this section will introduce UML, XML and the programming languages used in the configuration tools.

Python Python is a dynamic object-oriented programming language[23]. It is an interpreted language that has been designed with an emphasis on programmer productivity rather than optimized execution. The result is an easily readable language with very clean and concise syntax. It is very expressive in the sense that it communicates a program's intentions well in a simple way with a minimum amount of code. Like most interpreted languages, Python employs its own garbage collector to reclaim memory previously used by objects that are no longer active. Python is a dynamically typed language, meaning that the developer does not have to indicate datatypes of variables and that they are not really known before execution. With static typing, the developer has to declare the type of any variable before compilation, so that the compiler can be sure that the software is correct with regards to data types.

However, despite being a dynamically typed language, the Python philosophy is to ensure that errors surface for the attention of the developer and are not silently ignored by the interpreter. This makes a development cycle very fast as source code does not need to be compiled and errors are reported as soon as they appear during running of the software. Combined with unit tests it makes for a very powerful combination for

rapid software development.

One particularly important aspect with regards to its role in HLT software is Python's ability to be combined with other languages. A common usage of Python is to provide for easy scripting by embedding it into programs. The opposite is also possible; extensions can be written in C or C++ and used from Python, or bindings can be written for existing modules and accessed from within Python. Used in this way, Python can be seen as a glue-language, fitting together larger compiled modules into an application. This can for instance be used to do unit testing with compiled languages. For other languages than C and C++, there are implementations of Python in .NET (IronPython[24]) and Java (Jython[25]) which offers seemless integration into the respective languages.

Being an interpreted language, the main drawback of Python is naturally efficiency. In most cases Python will perform well enough and in cases where software needs to be faster, it is usually sufficient to optimize critical parts/code paths with better algorithms or reimplement them as extensions in a more efficient language.

C++ Although the configuration tools are written entirely in Python, the way Python and C++ are intermingled in the TaskManager makes it necessary to also have a good understanding of C++ to see the bigger picture. Based on C, C++ holds a strong position in software engineering as one of the most popular languages ever. It was among the first languages to bring object-oriented software development to the masses and is still widely popular despite the many new languages that have emerged, providing more advanced features.

Compared to Python, C++ is also object-oriented and supports multiple programming paradigms, but is otherwise quite different. First and foremost, C++ is compiled, it is a system programming language, characterized as being mid-level when it comes to abstraction. Further, C++ is statically typed and has no garbage collection. The software within HLT is predominantly written in C++ and C++ can also be seen as the preferred programming language throughout CERN due to its usage in projects such as ROOT and AliRoot.

Bash Command-line interfaces are usually driven by shells where users type in commands to the computer and observe the response resulting from the execution of the commands. There are several different shells, but the most commonly packaged by linux distributors is the Bourne-again shell or bash[26] (others are zsh, korn shell, c shell and numerous others). Commands can be collected in text files so that they can be executed all at once, like a script. Most shells also provide programming constructs, such as conditional statements and control loops to various degree, so that writing scripts becomes more flexible. Although some of the commands supported by a shell can be built in, most of the commands used are command line programs installed on the system. Most of these small programs are written with the unix dogma that they should only have one function, but that they should perform that one function well.

The configuration tools are, in addition to xml configuration files, also outputting bash scripts for various purposes. Some scripts are used for starting and stopping

the TaskManager, while others are used to check that everything is fine with the system before startup. A last type of scripts are used for cleaning up, for instance if a TaskManager fails or something else unexpected happens. As with the configuration files, these scripts are also created by inserting strings generated by the configuration tools into template files.

XML Extensible markup language is a subset of Standard Generalized Markup Language (SGML) created for storing data in a structured manner[27]. It is both human readable and easily parsed by programming languages. Since it is a pretty well-established standard, it is easy to find tools to operate on xml-files. It is therefore commonly used as an intermediate format for interchanging data betwen different software packages and programming languages. Listing 4.1 shows an example of a xml file.

The HLT software can serve as a good example of when it is beneficial to make use of XML, as XML is used as a common data container for both the TaskManager and the configuration tools.

The configuration tools are very much xml-driven as its structure is mostly shaped by the need to parse, process and create xml-data from the information within the xml-files. Several core xml-technologies are candidates that could be used for implementing xml-handling software. For parsing, there are a couple of different implementation strategies to choose from:

- **Simple API for XML (SAX)** [28] works by triggering functions defined by the developer as xml tags are discovered while reading sequentially through the document. This makes SAX a very fast and memory-efficient way of parsing xml although, to achieve its full potentional, skillful implementation is needed. In addition, because the xml data is transient to the program, the developer will be burdened with the task of keeping track of all the information the application will need for its operation.
- **Document Object Model (DOM)** [29] reads the entire document into memory and lets the developer access the xml-data in a more programmer-friendly fashion. It is possible to traverse the document and perform actions very much like a SAX parser would work, but it is also possible to use standardized methods such as XML Path Language (XPath) [30] for easy and intuitive access to xml-nodes. Methods for moving up and down the hierarchy is also available, so that one could ask for the parents or the children of a given node. The entire document is, in other words, always available (as opposed to SAX) to the program, so that references do not need to be kept.
- **ElementTree** is a Python api for a lightweigt, flexible and fast container object that is used to store hierarchical data structures[31]. There are two implementations, the original ElementTree is implemented in pure Python, while lxml is a Python-friendly implementation of the elementtree api using libxml2 and libxslt.

Both DOM and elementtree can also be used to build objects that can easily be serialized to xml.

Listing 4.1: Simple example of a xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<person_list>

    <person id="1" visible="true">
        <title>Sir</title>
        <name>
            <given_name>Arthur</given_name>
            <middle_name>Conan</middle_name>
            <surname>Doyle</surname>
        </name>
        <age>35</age>
        <occupasion>Carpenter</occupasion>
    </person>

    <person id="2" visible="false">
        <title>Miss</title>
        <name>
            <given_name>Laura</given_name>
            <middle_name></middle_name>
            <surname>Monneypenny</surname>
        </name>
        <age>65</age>
        <occupasion>Medieval Queen</occupasion>
    </person>

</person_list>
```

In the example in listing 4.1, DOM would parse the entire file into memory and enable the user to traverse its xml nodes along its edges. SAX, on the other hand would sequentially read the file, and provide the user with callbacks for each discovered tag that the programmer would then define the actions for. XPath could be used to easily access the nodes in DOM.

XPath XPath is a language that can be used to address parts of a xml document[32]. It uses similar syntax to paths as used by unix system and websites. For instance, in the example given in listing 4.1, all surname nodes can be retrieved by using a XPath expression as shown in the xpathEval method below. The code below also shows some of the possible operations that can be performed on nodes with the Python bindings to libxml2.

Listing 4.2: Xpath example usage

```
domDoc = libxml.parseDoc("filename")
```

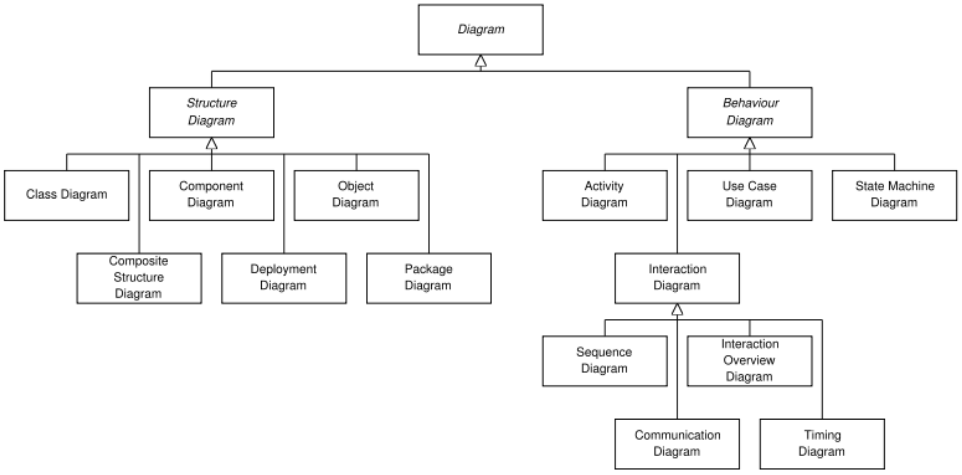

Figure 4.1.: All diagrams defined by the UML specification[5][6]

```
2 nameNodes = domDoc.xpathEval("/person_list/person/name/surname
      ")
3
4 for name in nameNodes:
5     print name.content        # print surname
```

UML UML is a visual language for specifying, constructing and documenting the artifacts of systems[6]. It is a general purpose language that can be used in any application domain and does not depend on any specific implementation platform. UML emerged as a unification of many ideas and methods in the late ´80s and early ´90s about analysis and design of software[33]. UML defines several different diagrams, that together form a graphical notation for describing different aspects of a system or a process[33], but UML also includes a meta-model; a model – usually in the form of a class diagram – that describes the notation itself. This is reflected by the fact that the uml specification is divided in two volumes: one for infrastructure that defines the foundational language constructs required for UML, the other for superstructure; describing the different graphical notations that can be applied to systems and processes[6].

UML diagrams can be divided in two main types; those which describe structure and those which describe behaviour and a there is a total of thirteen types. A diagram of all UML diagrams can be seen in figure 4.1.

The main idea of UML is to establish a common, well defined vocabulary – both visual and by means of words – to talk about systems and processes, the intention being to facilitate the communication of ideas and concepts. UML is therefore used

to some extent in this thesis, mostly elements of UML are used in diagrams when it makes sense.

4.2.3. Software, tools, libraries

In science related subjects, Free and Open-Source Software (FOSS) traditionally holds a strong position. Apart from being free – as in gratis – the flexibility and hassle free nature of FOSS makes it a perfect fit for the scientist who is concerned with solving problems, not being hindered by availability of source code and limited by licenses. For instance, if a Linux system does not fit a certain piece of hardware, the source code is freely available so that it can be changed and made to work. The CHARM card used in the cluster nodes is an example of Linux having been ported to smaller devices with a different CPU core.

Most of the software used when working with this thesis has been free or open source. Some of the more important packages and technologies are listed below.

linux[34] All activity regarding this thesis has been done on some kind of linux system. Mostly Gentoo, but also Ubuntu has been used. Linux is only the OS kernel itself with additional drivers, providing a means for software to communicate with hardware. The kernel is usually bundled together with other software in a distribution. There are many distributions each catering for different needs the user might have. Ubuntu will be used on HLT cluster. It is a derivative of Debian aimed at desktop users. One of its strengths is the excellent package system which it inherits from Debian. CERN has its own linux distribution called SLC. It is based on RedHat and uses a combination of rpm and yum for package management.

Gentoo[35] Gentoo is source based and can be considered a developers distribution since all developer tools are there almost by default. Gentoo also has consistent and intuitive tools for maintenance and is also well known for its extreme configurability.

ssh[36] Secure Shell is a tool for connecting to remote hosts in a secure manner. It provides the user with a remote shell where commands can be executed. Similar to rsh, but with increased security. In this particular setup it has been used together with kerberos to provide single-sign-on.

nx[37] Nomachine has developed a remote desktop solution that uses a secure connection over ssh and also compresses the data transferred so that it can be used over network with low bandwidth. It consists of nxserver and nxclient. Nxserver has also been used to provide access to gateway nodes in the HLT cluster for developers and engineers connecting from remote locations.

kde[38] kde has been used as desktop environment and its built in features is a great boon to this way of working remotely. For instance the kio-slaves makes it possible to open any remote file in any kde application by browsing to a link similar to this:

fish://flekke.homelinux.org/home/username/master.txt

fish is one of many kio-slaves (uses ssh for authentication/communication), so just like http is implemented as a kio-slave usable in everything from webbrowser to office applications, so is also ssh, ftp, svn (subversion), tar, smb, and the list goes on.

kate/konsole[39] [40] Coding has been done by using relatively simple tools such as an advanced text editor and console. The actual applications, kate and konsole, are both quick to use due to shallow keyboard shortcuts and deep integration into the desktop. For Python development, the only developer aid offered in kate is syntax highlighting. There is no code completion or debugging facilities integrated.

Command line tools[41] diff compares the differences between two files line-by-line and is often used when making patches. When working with files from different repositories, there is quite some effort needed to make things right. patch applies a patch created by diff to source code. diff and patch can help to ease exchange of code and make it easy to apply changes.

subversion[42] For version control, subversion has been used. It is simple to use and offers enough features to handle this project. With a subversion repository already in place, it is convenient to also use it for backup when writing thesis. Git or other distributed svc's might have been better, as merging code usually is easier with these tools.

kile[43] For writing the thesis itself, kile and kbibtex has been of great help when writing latex documents.

Umbrello[44] Is an UML modelling tool that can be used to draw uml diagrams. It has also limited capabilities as code generation tool for many languages and, additionally, it can also be used to create class digrams from source code. (Has been used to visualize old code, to draw uml).

kdiff3/kompare[45] [46] Is the graphical equivalents of diff. They are powerful tools for visualizing differences between files and for merging changes.

4.2.4. Technology

D-Bus D-Bus is a message system that tries to simplify the way applications talk to each other[47]. In other words it is a Remote procedure call (RPC) mechanism. Developed primarily for linux, the initial motivation was to enable desktop applications to talk to system level software, so that things like automatic handling of hardware can be implemented in a user friendly and consistent way.

Using a well defined interface with D-Bus enables the client software to be written in any language that has D-Bus bindings and no restrictions are made about the type of application. It could be GUI, Command Line Interface (CLI) or even web-based.

The Python bindings for D-Bus use decorators to highlight methods and signals that should be made available remotely over the interface. Below is an example of how a method could be defined in a D-Bus interface. The example uses Python.

Listing 4.3: D-Bus interface creation example

```
1  class NBusTaskManager(NBus):
2      # Decorators starts with a '@'.
3      @dbus.service.method("org.nbus.NBusService")
4      def getNodes(self):
5          # Return id string of nodes in dictionary
6          return nodes.keys()
7
8  # Initialization code for D-Bus.
9  # Interface name and object to be exported are set.
10 if __name__ == "__main__":
11     system_bus = dbus.SystemBus()
12     bus_name = dbus.service.BusName('org.nbus.NBusService',
           bus=system_bus)
13     object = NBusTaskManager(system_bus, bus_name, object_path
           ="/org/nbus/NBus")
14     reactor.listenTCP(g_port, pb.PBServerFactory(object))
15     reactor.run()
```

To access the method in the above code example is very simple:

Listing 4.4: D-Bus interface usage example

```
1  bus = dbus.SystemBus()
2  remote_object = bus.get_object("org.nbus.NBusService", "/org/
       nbus/NBus")
3
4  # Get the NBus interface
5  iface = dbus.Interface(remote_object, "org.nbus.NBusIface")
6
7  # RPC call on the remote object. Prints all nodes
8  print iface.getNodes()
```

D-Bus also provides simple scheme for security. Users and groups that should be allowed to use the different exported objects, methods and signals in the interface must be defined in a configuration file in the /etc/dbus-1/system.d/ directory.

Avahi Avahi is a implementation of a technology commonly called Zeroconf. It facilitates service discovery on a local network [48]. With Avahi an application can for instance be instructed to automatically discover instances of the same type as itself running on other hosts on the local network. For a client, this is done by connecting and listening to signals that are emitted when a given service connects or disconnects

to the Avahi daemon. Avahi also provides methods that can be used by a client to initiate service discovery. Avahi gives each service a unique id in the form of a string. The id string and other information about a service – such as IP address and port number – can be queried from Avahi about the host.

Twisted Is an event driven network engine written in Python[49]. Twisted can provide asynchronous network connection between nodes and remote objects over the local network (as opposed to D-Bus which operates only locally). The mechanism that provides remote objects is called Perspective Broker, which in twisted terms is called " a translucent reference to a remote object"[49]. This description speaks to Twisted's remote objects as being almost like local objects, just with an asynchronous twist.

Perspective Broker's implementation is based upon two central concepts[49]:

- serialization: by adding a few lines of code to existing classes, Twisted can turn them into objects that can be passed over the network. Even complex objects with references to other objects can be remotely accessed. This is achieved by serializing objects, sending them to their destination and reconstructing them there. Keeping close track of object id's is the key to make everything work.
- remote method calls: performing a method call on a local object will cause the method to be called on a remote object. The local object is called a RemoteReference and remote method call is done when calling the local object's .callRemote method passing the name of the remote method to be called as parameter.

4.3. Development considerations

4.3.1. Implementation guidelines

Underpinning the general implementation strategy is the belief that software development is most efficient with a well structured source code and proper testing in place. However, as implementing features are also an important part of this thesis, some guidelines will be defined, that will help balance (prioritize) the development, so as to achieve as many as possible of the goals set forth in this thesis.

In a perfect world, a developer would have the time to fully implement unit tests and perform all necessary refactoring before starting to implement features. This is however generally not the case and usually the developer will have to do a trade-off to meet his or hers objectives. So also with the work in this thesis. The effort to put in place testing and do refactoring will be done when it makes sense and the expected benefits are high. These will nevertheless be continous activities that (interchange) feature implementation activities.

Standard libraries As far as possible, functionality should not be re-implemented, but rather try to make use of standard libraries or external libraries with Python bindings. The advantages of re-use are: less code to maintain, relying on more heavily tested

software and Application programmer interface (API) that has to a larger degree been standardized.

Python is distributed "with batteries included", meaning that a comprehensive standard library is included in the distributed Python package. These libraries have lots of functionality readily available for the programmer.

Refactoring A well structured source code is a pre-condition for being able to work with the source code (i.e. adding features). If source is well structured, changes are easy to make and coding will be fast[50]. Refactoring takes time, but not to refactor takes longer in the long run and makes adding changes harder. To be able to accomplish all goals set forth in this thesis, there will not be a complete refactoring of the entire configuration tools, but refactoring will rather be done before features are implemented if necessary.

Testing Following an important agile principle, unit tests should be present in the development cycle. The most important benefit is that a developer can be confident that no bugs has been (re-)introduced when implementing features. Unit tests are also a pre-condition for refactoring, as otherwise the developer would have no way to verify that the changes made has not introduced any regressions. Regression, in software development terms, is the breakage of something that previously worked.

To write unit test before implementation might be hard with the current state of configuration tools; a refactoring might have to be done beforehand. Unit test will therefore be written only if not taking too much time for the given functionality. Before proper unit tests are in place, the only way to ensure that regressions are not occuring, is to check the output of the configuration tools against a reference output generated by version of the configuration tools known to work properly.

The test_tool program has been developed to perform this task as a compensation for missing unit tests. It takes a set of configuration files and executes the configuration tools with all of them and checks agains a reference output. If a feature is implemented and the output changes, then this new output will be made the reference output for which future revisions of the configuration tools are run against.

Optimization Generally, optimization should be done late in the development process, but as there are many things that could be changed, profiling can be used when deciding what to work on on first. For instance, the xml parsing proved to be slow, and replacing the used library with a faster one would anyway be beneficial. Profiling tests have been incorporated into the test_tool package and is run regularily as part of the daily development cycle. Looking at the results produced by test_tool can be comforting for the developer or indicate recent problematic code additions.

4.3.2. Automated build and installation tools

The source repository of hlt-software is located on the computer systems of Kirchhoff-Institut für Physik at the University of Heidelberg. As there is no easy way to give

commit access to users outside their academic institution, it was decided that development should be done in a local repository and that patches should be made against latest HLT-release. The idea is that patches are sent upstream and if accepted, they are included in the next release, otherwise they are discarded or improved until the point that they are accepted.

There is therefore a high incentive for getting the patches accepted, as otherwise one would have to maintain these outside the main repository and keeping patches in sync with new releasese can add significantly to the developer's workload.

For each new release, the source would have to be downloaded and the changes manually merged into the local repository. Then patches must be remade and applied on all test nodes before development can continue.

Continous integration Several of the recent development methodologies advocate the advantages of making continuous integration a part of the development process: The efforts required compared to postponing integration to a single final step are reduced and it allows faster development of cohesive software[51].

When a developer has finished implementing some functionality and is ready to check in the changes, then he/she should first sync with the central repository and build the entire system to see if anything has broken. If everything is ok, the changes can be checked in. If problems occur or conflicts arises, they must be resolved. When changes are checked in, the system should again be built on an independent build host to see if everything still builds.

An independent build host can be deployed to help with continuous integration. It will iterate through the entire process – from source checkout, building, testing and to deployment – continuously throughout the day. This improves the overall quality and makes developers more aware of deployment issues, which can in turn improve the user experiemce.

System wide installation/system integration Often when developing software, it is sufficient for the developer to use and test the software from the same environment from which it has been developed. In many cases this will be a home directory, which – because binaries and libraries are not in system-wide default locations – needs to be set up with lots of environment variables.

To replicate this specific setup may turn out to be non-trivial for a user (both for developers and for end-users), making the installation and usage unecessary complicated for the users. As part of this thesis, some effort has been put into having HLT software installed in system-wide standard locations.

Portage Gentoo's source based package system, portage, uses ebuilds which are basically a description of the steps needed to install a software package. It can in other words be viewed as a build system that happens to also be used as a package system. Portage has steps for fetching, configuring, compiling, testing, installing and finally merging packages. Fetching retrieves source code either by means of a source archive or from source repositories. Patches can be defined in this step and will be

automatically applied to the source. The installation step performs the installation procedure according to instructions included in the source distribution (if any, otherwise the ebuild writer will have to provide their own instructions) and directs the installation to a directory representing an image of a live filesystem. Merge, the final step, will simply move the files to their destination in the live filesystem[52]. Gentoo is often known as developer's distribution as developer tools are installed by default for building and installing packages.

The application of portage in this thesis is for exploring portage as a tool for integration, both at the software level within HLT software, but also for system-wide integration. A second path of exploration is with potentional users in mind. Together with layman and Source Code Management (SCM) software, portage can be used to provide reusable (across versions, on many architectures), distributed, automated installation instructions for HLT software.

There are several ebuilds written for building HLT packages and its dependencies. A slightly edited root ebuild was made for enabling pythia6 support. AliRoot, HLT framework and HLT components ebuild has also been written. To make everything conform to gentoo paths and build system, some patches have been made. There are hooks in portage to apply these automatically during package installation. Below is an example of how easy it is to install HLT software when it has been added to an ebuild overlay. This is considerably easier than the current installation instructions.

Listing 4.5: Layman usage: installing hlt software

```
1 # layman -a science              # add official gentoo science
      overlay
2 # layman -a flekke               # add private overlay used for
       HLT development
3 # layman -S                      # sync all overlays
4 # emerge -ap alice-hlt           # install alice-hlt and
      thereby all it's dependencies
```

4.3.3. Development environment/setup

The development work involved with this thesis has been done on private nodes, geographically localized quite far from the premises of Institutt for Fysikk og Teknologi where the assigned workspace has been. A laptop is used to remotely connect to the nodes by means of ssh[36] and nx[37] client/server software.

There are several advantages to being in complete control of the development system. First, is the familiarity resulting from choosing the preferred operating system and software. Being familiar with the system will naturally result in increased efficiency, but more intangible effects, such as feeling "at home", can also have a significant impact on productivity.

Developing in preferred environment is an idea borrowed from webprogramming. By first coding for the preferred browser and then adding the changes to make it work

with rest of browsers, you get a focus on functionality and can deal with implementation issues in the different browsers later when a solid design foundation has been laid. This translates well into the domain of software development, as there might be several target platforms. Putting the programmer in a familiar environment will let the programmer focus on implementing functionality and making a solid design without having to consider integration details until later.

To be able to change the setup at any time without having to take into consideration other users needs or being dependent on system administrators, dramatically reduces turn around time for implementing changes to the setup.

The disadvantages with such a setup are quite clear. Because of the distance, it will take a lot of more time and effort to repair any failing part of the setup. The accessibility of such a system also depends on several more factors, such as the providers of internet connection and power. Although losing contact with nodes has high impact, experience has shown that the chance of this happening is remote. Therefore, in this particular case, the advantages are considered to outweight the disadvantages. To further minimize the risk of losing contact with the nodes, the policy has become to only perform critical upgrades to the system when in close proximity of the machines.

The setup might also differ quite a bit from the intended implementation site. Migration and integration could therefore become unnecessary complicated. To remedy this, an independent build host could be deployed. With properties similar to the application domain (i.e. virtual ubuntu installation), it would be in accordance with the agile practice of using independent build hosts to continuously do integration testing.

5. Solution

All tasks will be presented in the form of user stories in section 5.1. The user stories will introduce tasks that are new to the discussion in this text. These are tasks that have been added either by the customer or developer after the initial problem was described. On the developer's part, tasks have been added as a result of following the guidelines outlined in section 4.3.1. Tasks added by the customer are either changing requirements or request for new features, both of which should be expected to happen in any project, but the hectic and resource-strained nature of HLT makes it particularly relevant.

For every user story, there will be a matching solution in section 5.2.

5.1. Task break down

This section will consist of a mix of user stories as they are reformulated from tasks to be resolved and user stories written along the way for formulating code improvements in the spirit of agile development and constant source code improvements.

5.1.1. Improve XML parsing code

During the initial rounds of becoming familiar with the source code, a dependency on PyXML[53] was spotted. A wrapper library, XMLReader.py, had been created to provide helper methods for easier handling of XML and used modules from PyXML for its functionality. The configuration reader for MakeTaskManagerConfig.py was also using this module. See figure 6.1 for a visual representation.

Performance profiling showed that the XML parsing code was responsible for most of the computing time spent by the configuration tools. PyXML is an entirely Python based XML parser, and will therefore not be able to compete with a well written C or C++ parser when it comes to speed, due to the overhead imposed by Python being an interpreted language. Furthermore, compared to some packages, PyXML have weaker support for some of the features which makes working with XML easier.

The original parser was based on a SAX [28] implementation, which is considered to be fast and memory efficient, but complex to write. An alternative would be to base the parser on DOM [29] and XPath [30]. Such an approach would generally need less code to accomplish the same task, while still being more intuitive to read than the event based SAX parser.

More in depth information about xml technologies and programming languages can be found in section 4.2.2.

User story 1: Improve efficiency of XML parsing code in configuration tools.

User story 2: Simplify source code of XML parser to improve expressiveness and to increase readability and maintainability.

5.1.2. Usability improvements chain operation

During development, some ideas have been shaped more or less subconsciously about improvements as to how a configuration chain can be operated. This also has the end users in mind. These elements have been collected as a description here with accompanying users stories.

MakeTaskManagerConfig.py has a lot of optional arguments it can be passed. These makes it easy to adapt the software to different systems, where paths for instance can differ. To remember these and avoid retyping them everytime a command needs to be run, the operator typically puts the commands in bash scripts, which in turn is used for execution. A more general and portable solution would be to put the very same information in a XML file, strictly for site specific information. This information includes paths to output, framework, run and several other directories, as well as start of port range for component communication and program paths. This configuration file could be read by configuration and run-control scripts to retrieve information about the environment it is running in.

Operation of a given chain should be as easy as possible for the end user, who is only interested in testing their own component or the data produced by the chain. Operations made available to the end-user should be limited to activating one of several prepared configurations, start and stop of the chain. Optionally, commands for starting a GUI for more directly manipulating running processes and a command for starting a GUI application for observing events, could be made available.

User story 3: Extract site specific information into a XML file that can be used as a common source of site specific information for configuration scripts.

User story 4: Make a script whose purpose is to list available configurations to the user and to configure a site with the configuration that the user chooses. It should use site configuration from user story 3 for retrieving needed site specific information. Suggested name: hltConfigure.py. Example interaction:

Listing 5.1: Example execution hltConfigure

```
1 # hltConfigure --list
2         1 - pp_test
3         2 - Pb-Pb
4 # hltConfigure 1
5         Configuring nodes with configuration: 1 - pp_test...
```

User story 5: Make two scripts that has the purpose of starting and stopping a configured chain, respectively. The scripts should use the Python module TMControlInterface for interacting with a started TaskManager to get a chain to running state or stop it. The start script will have to start the TaskManager before interacting with it and the stop script will have to stop the TaskManager after instructing it to stop all its processes. Should use site configuration from user story 3 for retrieving needed site specific information. Suggested names: hltStart.py, hltStop.py. Example interaction:

Listing 5.2: Example execution hltStart and hltStop

```
1 # hltStart
2         Starting configured chain: pp_test
3         .
4         .
5 # hltStop
6         Stopping configured chain: pp_test
7         .
8         .
```

User story 6: In order to make the set of commands complete and coherent, make two scripts. One that starts the TaskManager GUI and one that starts the AliHLTGUI for event observation. Should use site configuration from user story 3 for retrieving needed site specific information. Suggested names: hltTMGUI.py, hltAliGUI.

5.1.3. Distributed configuration creation

Generating all configuration files for an entire HLT analysis chain takes a substantial amount of time. Thousands of files will be created. Creating all these files takes time even on fast machines. For a very simplistic analysis chain covering a single side it takes approximately 1 minute on an Athlon XP 2500+ machine with 1 GB of memory (see appendix A). A more elaborate and realistic chain covering both sides, would take considerably longer. With all the computer power that is available, it is more than sensible to try to make as much use of the available resources as possible. Therefore it would be useful if the task of configuring the HLT nodes could be done in a parallel distributed manner.

Two scenarios can be envisaged for the intended use of such a solution. First, one could let every node create its own configuration files. A second possibility would be to restrict configuration generation to a small set of nodes, scaling arbitrarily up to a point where the creation is sufficiently fast.

Another element to keep in mind is that depending on storage technology used in the HLT cluster, distributing configuration generation could further improve performance and reduce strain on the local HLT network. The HLT network uses Andrew File System (AFS) as distributed filesystem. All nodes could read common master configuration from AFS and store it on their local disk. This solution avoids much

network traffic caused by writing to AFS and saves AFS file server for lots of read and write operations.

User story 7: Modify configuration software so that it becomes possible to restrict the file generation to files that are only relevant for a single machine in a configuration. An optional argument, singleNode, should be added to MakeTaskManagerConfig. The argument passed to this argument defines which node configuration files should be made for.

5.1.4. A mapping program

The configuration file that MakeTaskManagerConfig.py takes as input holds all information needed to create run-control scripts and TaskManager configurations for nodes participating in a certain configuration. The input configuration has basically two sections: a list of nodes that is a mapping of node identificators to real hostnames and a process list. Each process list item contains fields for the command to be executed for the given process, but also information about which process(es) is its parent(s). Processes are listed sequentially in such a way that parent processes are defined before its children (see appendix B for an example). Creating the process identificators and inserting them where they belong – so as to establish the relationship between them – is a manual process not aided by software. This is also true for the mapping of node identificators to hostnames.

A configuration file can be thousands of lines long, so creating them by hand can be an error-prone and mundane task for the user. Therefore, to further automate creation of configurations, a program should be created that automates the creation of master configuration files that are used as input to MakeTaskManagerConfig.py. It should use seed files in XML format for input. There are two user stories:

User story 8: The mapping program should map node identificators to real hostnames based on definitions in a seed XML file.

User story 9: The mapping program should create process identificators and properly map them to processes in such a way that the relationship between processes are established in the resulting master configuration. This mapping should be defined in the same seed file as for user story 8.

5.1.5. Avoid recompilation of Python bytecode

The first time a Python program is run, a bytecode file is saved so that next time the program is loaded into memory, loading time can be shortened by loading the byte code directly instead of parsing the program file itself. This is done transparently as far as the user is concerned, who should only notice a slight decrease in the time it takes for the program to load. The configuration tools are largely based on mixing template files of Python code into configuration files that are in the end parsed by the

TaskManager upon startup. When the configuration files are parsed, the Python code is also loaded into memory by the TaskManager. This organization does not allow for taking advantage of faster loading of precompiled Python code.

User story 10: Find a way to take advantage of Pythons bytecode features.

5.1.6. Repeated creation of configuration objects

Initially, configuration elements are entered into a XML file and stored there for input to the configuration tools. Then, when the XML configuration file is parsed by the configuration tool, every element is made into a Python object, the software performs its operations and outputs configuration elements as a XML file again. The startup scripts loads the TaskManager with its proper configuration. Now the node specific configuration files are parsed again and this time its elements are made into C++ objects.

User story 11: Investigate to see if it is possible to avoid some of the steps mentioned above.

5.1.7. Explore different approach

Some of the requirements for the work with this thesis presented a couple of problems that may be hard to resolve with the existing architecture of the software. After getting to know the source code it might be a good idea to do a quick prototype to explore a different approach.

The user stories will naturally be broad and open to allow for experimentation, but as a starting point, cases that are hard to solve with the current architecture should motivate the direction of an implementation. There are two cases that stand out in this regard and that is reducing repeated creation of configuration elements and avoid recompiling Python bytecode. Additionally, as the configuration tools are to be deployed in a large cluster, it makes sense to utilize the computer resources available and distribute the task of creating the configuration files. The locality implied by such a solution might also help simplify the implementation.

The general idea is to distribute configuration objects over the network and have the receiving host create their own configuration and run-control files. This in contrast to having the configuration tools generate all configuration files on one node, made available to the other nodes over a distributed filesystem, and TaskManagers by using remote shell.

User story 12: Write a prototype with a different approach. Specifically, one that tries to be more object oriented and distributed.

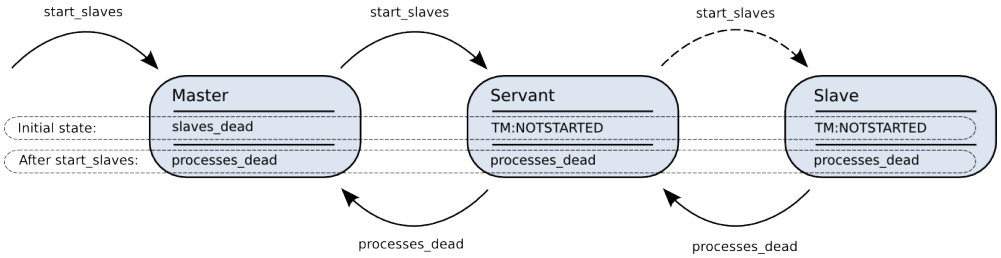

Figure 5.1.: State propagation in TaskManager hierarchy

5.1.8. Finalize servant/node group implementation

First, when a chain configuration is started, there is only the master TaskManager running. All other TaskManagers are started from the master. Typically a client application will connect to the master and assume control over the entire chain by executing commands on the master TaskManager. The state of all other TaskManagers are at this point TM:DEAD, meaning TaskManagers are not started yet. The master itself is in state slaves_dead. To start a chain, the client starts by issuing the start_slaves command on the TaskManager. When this happens, the master passes this command on to all its slaves, (slave TaskManagers as well as servant TaskManagers). The purpose of issuing a start_slaves command is for the master TaskManager to go to processes_dead state. This state means that all TaskManagers (master, servant, slave) have been started and that the chain is ready to start the analysis components (processes).

When a servant TaskManager is started, it does not start the slave TaskManagers that are part of its node group, as would be the intention. When starting up, the servant should start all slave TaskManagers that are part of its node group. The StartupAction tag for the TaskManager configuration files would be the natural place to implement this functionality.

The dotted arrow in figure 5.1 shows the step that is missing.

User story 13: The servant TaskManager should start all slave TaskManagers (servants (representing nodegreoups) as well as slaves) that is part of its node group.

User story 14: It should be possible to define additional servants to a node group. The configuration tools should generate files that allow for dormant servants to replace other possibly failing servant TaskManagers.

5.2. Chosen solutions

As this thesis tries to resolve several diverse problems, their solutions will be presented as smaller independent pieces, rather than a single large picture. However, as part of

a more general solution, implementation guidelines that are intended to help guide the implementation, are defined in section 4.3.1.

5.2.1. Improve XML parsing code

The Python bindings for libxml2 has been chosen for implementing the user stories for improving xml parsing code. Since it is written in C, it has the potential to speed up parsing considerably and the support for XPath expressions should prove to be a good opportunity to re-implement the code in a much more clean and intuitive way. The combination of DOM and XPath will form the basis for the solution and the original SAX design will be abandoned. The ElementTree implementation in lxml (discussed in section 4.2.2) would also be a good candidate, but the capabilities of the Python bindings included in libxml2 should be sufficient for the task, while being closer to the native library (less overhead).

5.2.2. Usability improvements chain operation

The site configuration file will be a simple xml file with a only one level of nodes below the root element. This xml structure will be translated into a Python dictionary, where tags are translated into keys and the content of the tags are the value of the dictionary element. Implementation of the helper programs described in the user stories should be straight forward. libxml2 will also here be used to parse the site configuration and Python will be used as implementation language. Correct usage of the Python module TMControlInterface should be enough to make the start and stop scripts work. The configure script will know where to find configurations by looking up information in the site configuration.

5.2.3. Distributed configuration creation

The feature of this task will be implemented so that if the value passed to the singleNode argument is omitted, it will default to the host the program is run on. Introducing this argument in MakeTaskManagerConfig will restrict creation of configuration files to the node specified. To create the configurations for all nodes in a chain configuration, MakeTaskManagerConfig will have to be run with the singleNode argument on all nodes. In order to achieve this a simple Python program (create_node_configs.py) will be created that uses the os.system module of Python to execute command line programs. The program will use the site configuration file to retrieve information about the setup and get a list of all nodes from the chain configuration file. It will then iterate over all nodes participating in the chain configuration and execute a prepared MakeTaskManagerConfig command with a remote shell command such as ssh or rsh.

5.2.4. A mapping program

A seed xml file will be established which will be the input to a Python program create_master_config.py. It will contain sections for defining a master, node groups,

tasks, process levels and template strings. Together they will be able to construct chain configuration files in the format that MakeTaskManagerConfig accepts. The crux of this piece of software is that there should be no identificators needed to be entered manually to define relationship between processes. The participating elements will be automatically enumerated and identificators will be constructed from these and identificator prefixes given in the seed file.

5.2.5. Avoid recompilation of Python bytecode

To avoid recompilation of Python bytecode, functionality can be pulled out of the templates and configuration tools and put into modules almost like a library. There would then have to be an import statement in addition to the existing KIPTaskMan (the module made available to Python programs for TaskManager control) and the module would have to be in the Python path. This is one task that might be more easily solved by a different approach.

5.2.6. Repeated creation of configuration objects

This task might also be more easily solved by a different approach.

5.2.7. Explore a different approach

For the sake of discussion in this text, the prototype has been given the name NBus. NBus represents the more novel aspects of this thesis as the original tasks were to a larger degree about implementing suggested solutions rather than finding a solution to a problem.

For implementing the user story given in this task, D-Bus will be used to create the interface that client applications must use to interact with the NBus daemon. Only NBus daemons running on hosts that will provide interaction with client software will need to be equipped with this D-Bus interface.

Avahi will keep track of nodes hosting the NBus service. Upon startup NBus will register itself with Avahi and then use Avahi to discover hosts on the local network that also runs the NBus daemon service. From then on, by listening to the appropriate signals, a NBus daemon will be able to maintain an up-to-date arrary of NBus enabled hosts almost for free. Avahi will be accessed by using its Python D-Bus bindings.

Finally, Twisted will provide a solution for performing operation on remote objects; NBus will keep a remote reference – retrieved by the addresses provided by Avahi – for every participating node. So, in general when a client uses the D-Bus interface of NBus to call methods, NBus will use its Perspective Broker remote objects to call appropriate methods on all the relevant nodes. The result will be collected by the NBus instance on which the client initially called the method. In the end, the client has to listen for a given signal emitted by NBus when it has finished collecting all responses within the twisted framework for its final response.

A more detailed description of the different parts that will be used in this prototype can be found in section 4.2.4.

5.2.8. Finalize servant/node group implementation

The first user story will be completed by adding node IDs in the StartAction tag of the chain configuration file in the same way it is done in other action tags. Then iterating over these and issuing a start_slaves command should be enough to bring all TaskManagers to processes_dead state.

The second story should be implemented by letting it be possible to add more than one master to a node group. These additional servants need to be added to the master configuration. Then, when the master TaskManager is started, only one of them will be started, but in case a servant fails another servant assigned to the given nodegroup should be started and assume responsibility.

6. Implementation

This chapter will continue the theme from previous chapters of presenting the tasks seperately and treated independently in their own sections. At the end of this chapter there will be a summary assembling all the threads and highlighting how the overall structure has changed.

Although it was realized at an early stage that a major refactoring would be beneficial for the rest of the work with the source code, it was nevertheless not started properly until very late in the process. The wish to stay more or less in sync with upstream, was one reason that made it more difficult to introduce major changes to the source base.

As a result, many of the user stories have been implemented before any refactoring and would therefore have to be refitted and merged together to become a consistent set of tools. This process has for the different solutions reached different stages of completeness and will not be finished by the time this thesis is submitted. An indication will therefore be given about the status for each of the user stories.

6.1. User stories

6.1.1. XML parser improvements

The class XMLConfigReader was extracted from SimpleChainConfig into its own source file. The dependency on XMLRead was removed as this class became redundant with the introduction of libxml2. The rest of dependencies on XMLRead in HLT software was also removed, so that XMLRead could be removed altogether. The diagram in figure 6.1 shows the original structure of configuration tools with regards to its xml parsing elements.

Python bindings for libxml2 were used to rewrite XMLConfReader.py and replace the functionality in the source code using PyXML. The resulting code was much shorter, while still clearer and easier to read (due to using Xpath and DOM). Being a C library, libxml2 outperformed PyXML by a factor of two when tested (see table 7.1).

For comparison, the structure after refactoring and new implementation xml parsing code is shown in figure 6.2.

Status: The improvements made to XML parsing have not been affected by recent refactoring. The new XMLConfigReader is part of the main HLT package and the two first user stories are therefore considered to be done.

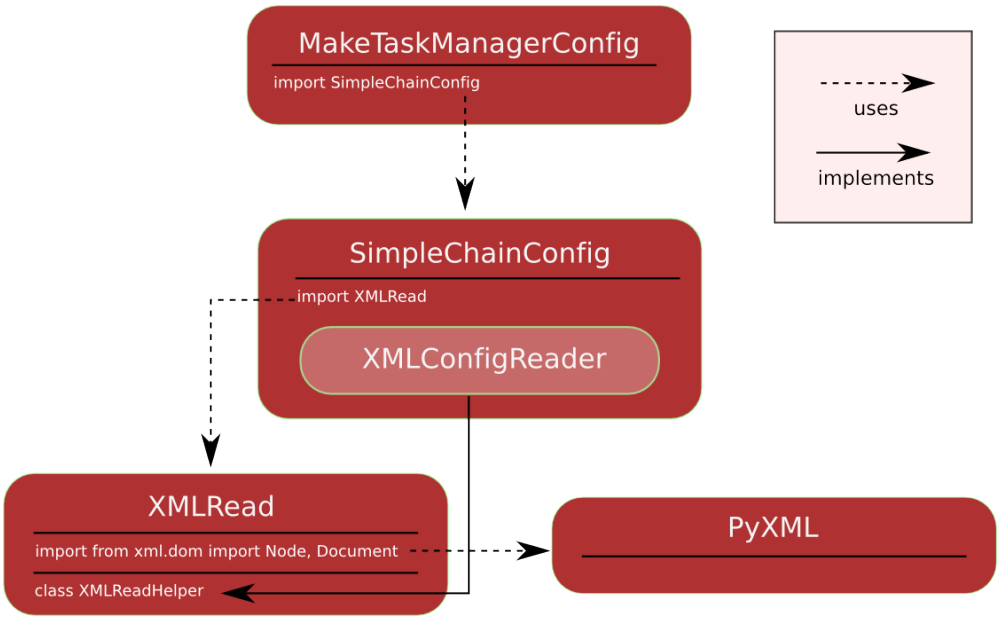

Figure 6.1.: Initial structure of xml parsing elements in configuration tools.

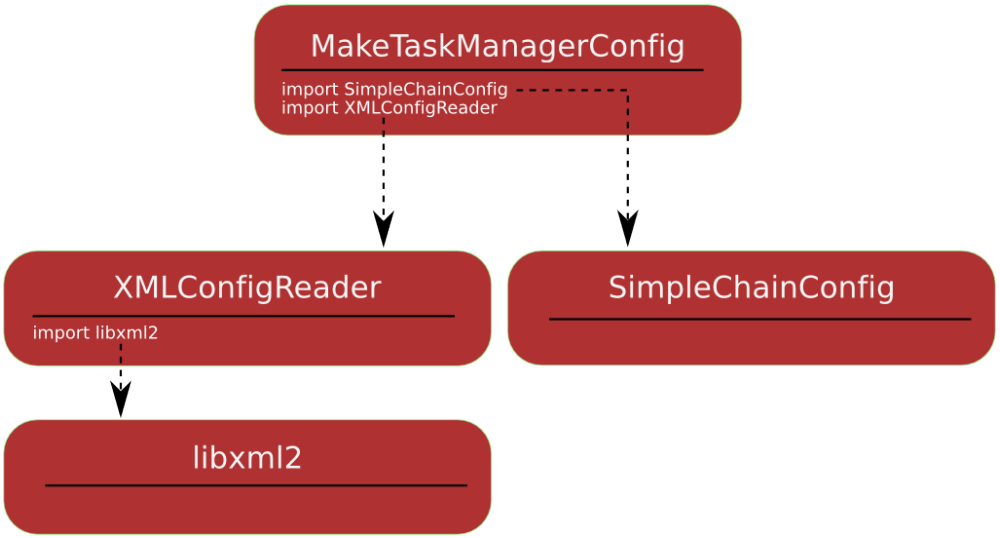

Figure 6.2.: Structure after refactoring and new implementation of xml parsing.

6.1.2. Usability improvements chain operation

After implementing site configuration in user story 3, the rest of the configuration tools were modified to also make use of it. Settings are brought into the software in very much the same way as illlustrated in example 4.2.

The program in user story 4 uses a directory defined in site_config.xml for finding where chain configurations are stored. hltConfigure.py can list all these configurations to the user and make a configuration active upon a user request. Further features that could be implemented is functionality for listing the currently active configuration and checking a configuration for correctness.

User story 5 was implemented by starting the master TaskManager from Python by using os.system. This module starts programs on the command-line. hltStart.py has to wait a couple of seconds for the TaskManager to be brought up. Then a connection is made to the master TaskManager via the Python control interface. From this point a sequence of state queries and commands are used to bring the chain from processes_dead state to running state.

For hltStop.py, the process is the reverse of hltStart.py. Commands are used to take the chain to processes_dead state, but in the case of stopping the chain, it is possible to use the quit command, so that Python does not need to explicitly kill the master TaskManager process.

The original MakeTaskManagerConfig program needed – as can be seen in the example below – a lot of parameters passed to it when invoked. This is not necessary when using create_node_configs.py, as this information is now stored in an xml file and retrieved from there by the configuration tool. Since xml is stored in a plain text file, it is still quick to make changes, but there is a lot less to type when executing commands.

Listing 6.1: MakeTaskManagerConfig.py example

```
1 MakeTaskManagerConfig.py -usessh -masternode fep_00_00 -
    taskmandir /opt/hlt -frameworkdir /opt/hlt -prestartexec /
    home/hlt/bin/prestart.sh -emptynodes -production -config
    chain_config.xml
```

Status: User stories 3, 4 and 5 are implemented, but 4 and 5 might need to be updated because they have not been touched for a while. User story 6 has not been done, but is considered to be trivial to implement.

6.1.3. Distributed configuration creation

Was implemented by introducing conditional constructs that executes the outputter methods only for the node passed as argument to the singleNode parameter of MakeTaskManagerConfig.py and do so only when singleNode is used. If no arguments are passed, localhost is used by default. To make this feature run on an entire cluster, a new Python program was created that iterates over all nodes participating in a configuration and executes MakeTaskManagerConfig.py remotely – using remote shell – with

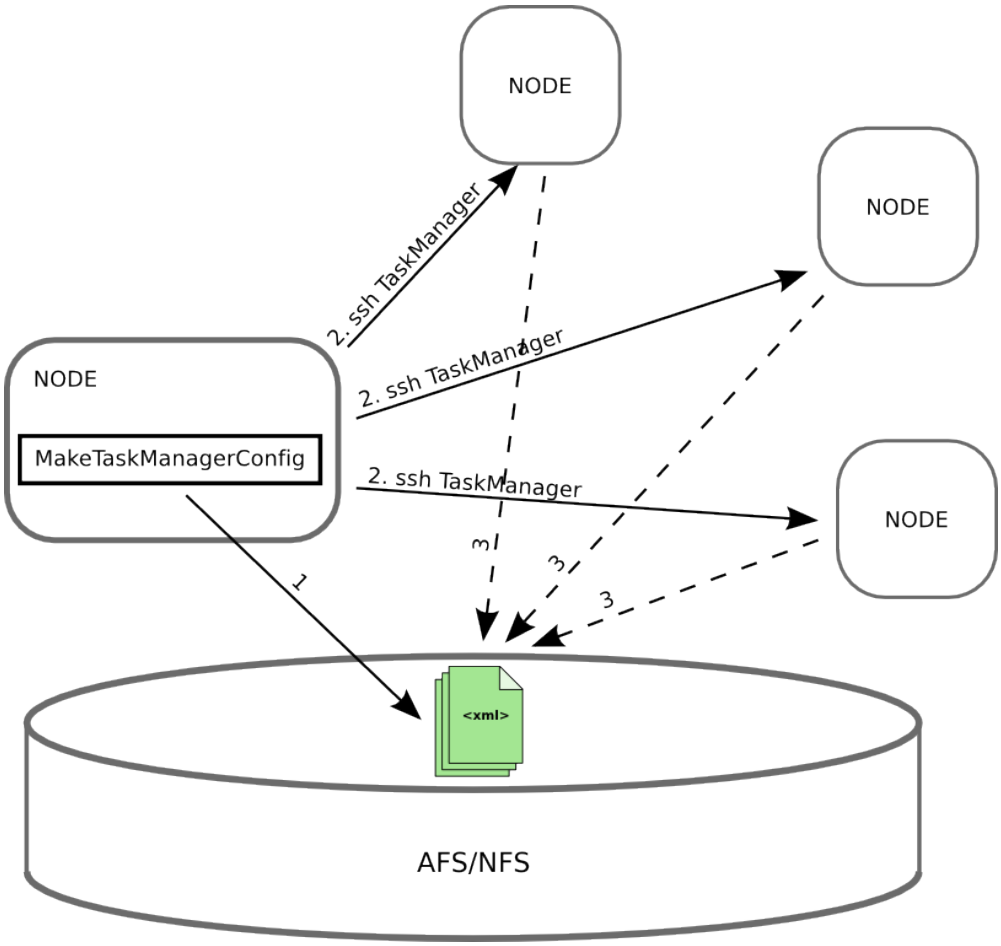

Figure 6.3.: Configuration tools in non-distributed mode

the proper arguments to singleNode. This program is called create_node_configs.py and also uses site_conf.xml (user story 3) for site specific information.

The difference between the two modes can be seen comparing figure 6.3 and figure 6.4.

Status: User story 7 is implemented, but will need to be re-implemented after refactoring.

6.1.4. A mapping program

As explained in the user stories, it is quite tedious to create chain configurations. Since there was a need for creating several different chain configurations to do testing, the

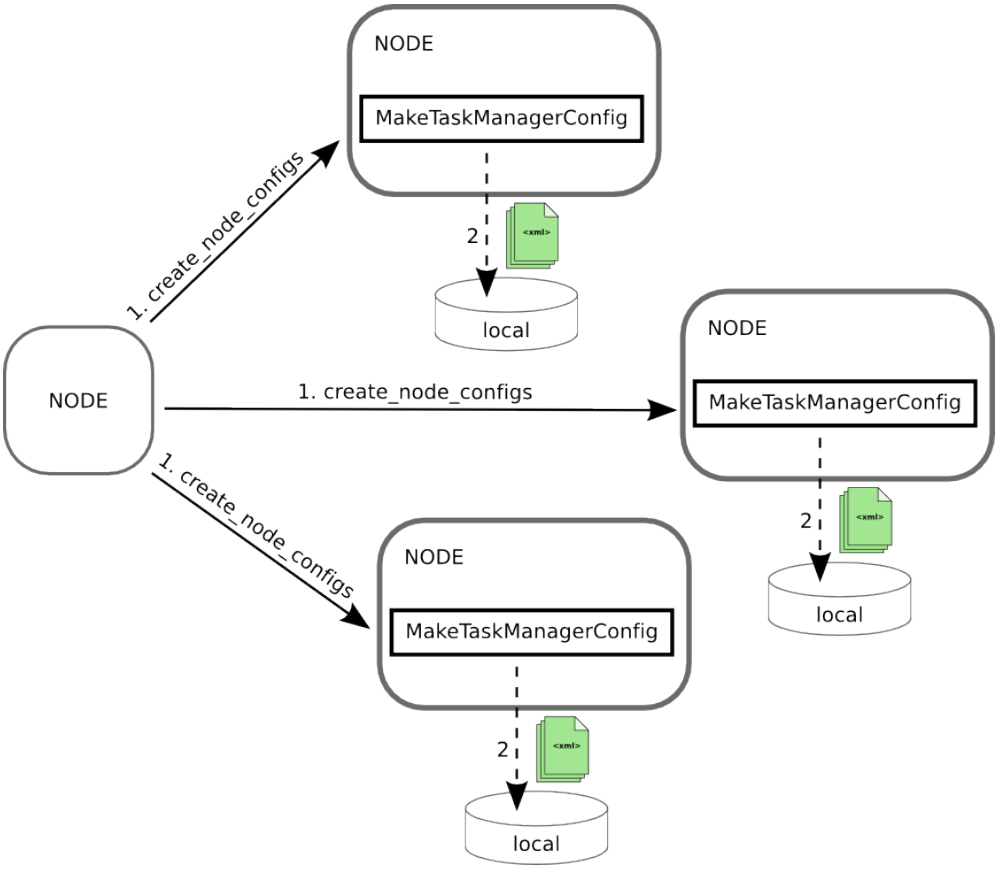

Figure 6.4.: Configuration tools in distributed mode.

create_master_config.py program was written. Its implementation is best explained through an example of how it operates.

As an example, consider this segment from the task list section of a seed file:

Listing 6.2: Task seed configuration example

```
<task typeId="fep" quantity="3">
    <processlist>
        <proc typeId="FP"/>
        <proc typeId="CF"/>
    </processlist>
    <hostlist>
        <host name="katten"/>
        <host name="knutn"/>
        <host name="kallen"/>
    </hostlist>
</task>

<task typeId="trk" quantity="1">
    <processlist>
        <proc typeId="TR"/>
    </processlist>
    <hostlist>
        <host name="fuzzy"/>
    </hostlist>
</task>
```

The first task defines a front end processor (FEP) where each node has has two processes: a file publisher (FP) and a cluster finder (CF). The hostlist defines three hosts (a node group could be given instead). Quantity tells create_master_config.py how many instances of the task that should be created. Here there are three tasks and these will be assigned to the three hosts. When a host is assigned a task, it is given a node mapping. A host can only have one slave TaskManager running, which means there can only be one task for each node. create_master_config would here construct identificators so that processes would be called FP_0_0 - FP_0_2 and CF_0_0 - CF_0_2, while nodes would be called fep_0_0 - fep_0_2.

Next, one tracker (trk) task is defined with one host. Again, the resulting identificators would be TR_0_0 for processes and trk_0_0 for the node.

Process levels are used to define the actual mappings:

Listing 6.3: Process level seed configuration example

```
<processlevels>

    <level typeId="FP" templString="filepublisher"/>
    <level typeId="CF" templString="clusterfinder">
        <parent parentType="FP" quantity="1"/>
```

```
6       </level>
7       <level typeId="TR" templString="slicetracker">
8           <parent parentType="CF" quantity="3" />
9       </level>
10
11  </processlevels>
```

The first file publisher (FP) process will be assigned to the first front end processor (FEP) node (and so on), but no more mapping is needed since the filepublisher process does not have a parent process. The clusterfinder process, on the other hand, has one parent (quantity=1), which is of type filepublisher (FP). Similar to process to node mapping, the first available filepublisher vil be assigned as parent to the first clusterfinder and so on. Each level also defines a template string (templString) which is the last piece of the puzzle.

Listing 6.4: Template string seed configuration example

```
1   <templstring id="clusterfinder">
2       <Proc ID="%s" type="prc">
3           <Cmd>AliRootWrapperSubscriber
4               -componentid TPCClusterFinderPacked
5               -componentlibrary /home/aliroot/AliRoot_v4-04-10/
                    HLT/lib/libAliHLTTPC.so
6               -componentargs "pp-run rawreadermode offline"
7           </Cmd>%s
8           <Node>%s</Node>
9           <Shm blocksize="125k" blockcount="12" type="
                bigphysarea" />
10      </Proc>
11  </templstring>
```

The template string above is for a clusterfinder. The process level CF links (in this example) to this template string by the id: clusterfinder. In the example the string sequence "%s" appears several times. This is a Python symbol meaning that a string can be inserted into that position in the Python string. The necessary identificators created above are inserted where they belong into this template string. There is a attribute for the proc tag that receives the process id. The designated node for the process is inserted in the node tag and optional parents are listed above the node tag (therefore, the parent tag is not there before any parent is inserted).

Status: User stories 8 and 9 works with the current implementation, but will have to be continuously updated to fit with changes made in configuration tools.

6.1.5. Avoid recompilation of Python bytecode

Provided motivation for NBus implementation, but has not been implemented.

Status: Not implemented.

6.1.6. Repeated configuration creation

Also not implemented, but provided motivation for NBus prototype. The addition of create_master_config.py could actually be seen as creating yet another transformation for configuration elements. It is the hope though, that refactoring will make obvious ways in which steps can be merged and code can be simplified.

Status: Not implemented.

6.1.7. NBus prototype

The main idea behind NBus is to make it easy to send objects over the network and manipulate them remotely. At the heart of NBus is Twisted, which provides asynchronous RPC functionality. By running a NBus daemon on every node, Python configuration objects could be distributed over the network and configuration files would be created locally by NBus, thus distributing the workload and possibly avoiding some of the translations configuration elements currently has to go through (as mentioned above). While being a very different approach, it has nevertheless been a goal to reuse as much code as possible.

Other benefits would be that this architecture would make it easy to write end user software in an object oriented way with Python instead of using bash scripts. Setup of passwordless ssh/rsh would not be needed. Existing code could probably be re-written to take advantage of locality and thus be simplified.

All running instances of NBus are equal, with the exception of the instances that provides an interface to client applications. NBus builds on several emerging technologies for the Unix platform to build as much as possible on well tested code that delivers lots of functionality in C and C++ libraries for easy access in Python. There are three key technologies NBus makes use of: D-Bus, Avahi and Twisted. An overview of the system and its components is shown in image 6.5.

A sequence diagram for a typical session with NBus is shown in figure 6.6

The first prototype of NBus was equipped with heavily modified classes from SimpleChainConfig. It was tested in a very simplistic setup, with a simple FilePublisher as the only analysis component and all processes running on the same node. In this simple setup, NBus was able to configure the node and start and stop the configuration chain.

Objects are as planned transferred to the "remote" hosts and then configuration files are written. When this is done, it is possible to start the chain.

A second round of fleshing out the implementation tried to make the NBus framework more general and so two components was created as test cases (Console, Info). These test cases were tested with setups consisting of three nodes. It is therefore believed that all elements of the suggested implementation in user story 12 have been demonstrated to work.

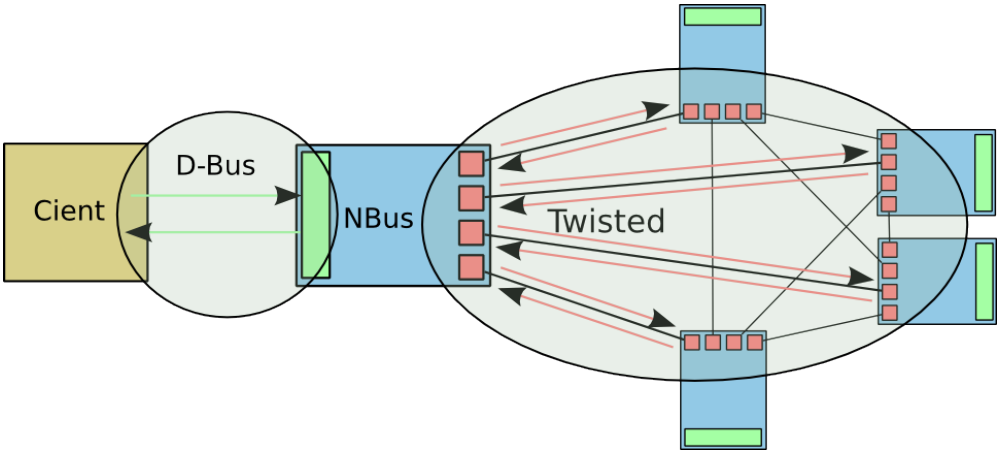

Figure 6.5.: Overview of technology used in NBus.

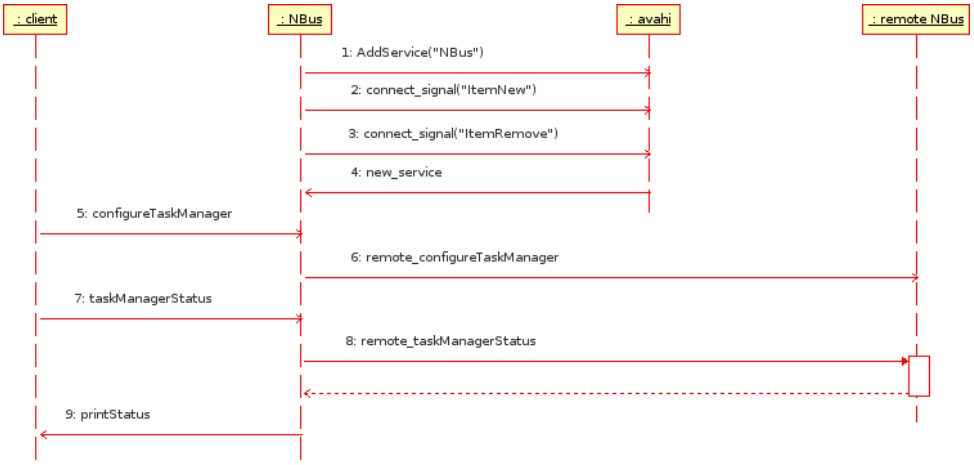

Figure 6.6.: Sequence diagram for NBus.

Status: Proof-of-concept found to be working. There is of course much work left to implement the solution properly so that one would know for sure that the NBus would be able to handle the task.

6.1.8. Servant/node group

User story 13 was implemented as suggested in section 5.2.8. At this point refactoring had been made part of the development process and was put into use when needed. Some of the changes (building a xml doc that gets written to file instead of writing xml strings one by one to file) made during implementation of user story 13 would be nice to have completed throughout the source code before starting user story 14. This is not done yet and user story 14 will therefore have to be finished later.

Status: User story 13 finished, but user story 14 has been put on hold until refactoring is done.

6.2. Structural changes

As the work with this thesis took shape and progressed, there has been a slight change in the main theme of implementing node group and servant. While still a large part of the thesis, the servant implementation has been accompanied by tasks that address usability, testing, distribution of workload and not the least refactoring.

All in all, the internal structure of the original source has not changed much during development. Most of the tasks have added functionality in new programs while making use of the existing code. The notable exception being the rewritten XMLConfigReader and NBus. However an ongoing refactoring – that will not be done in time of the deadline for this thesis – will change much of the inner workings of MakeTaskManagerConfig and SimpleChainConfig.

Figure 6.7 highlights the difference between the original software and the current situation.

As refactoring has progressed, there has been a move from using MakeTaskManagerConfig.py and executing it on the command-line via Python modules towards using objects directly from SimpleChainConfig. The addition of the site_config.xml file, made this even easier as the configuration options from it can conveniently be collected in a simple structure (dictionary) and passed around in the program.

6.3. Contribution

The NBus implementation is inspired from the desire to easily distribute and manipulate remote objects. It is a novel implementation in the sense that technology were chosen and put together to achieve this goal without knowingly being inspired by similar existing software. Working with the original configuration tools has been more about learning to use the provided libraries and use these to implement the desired

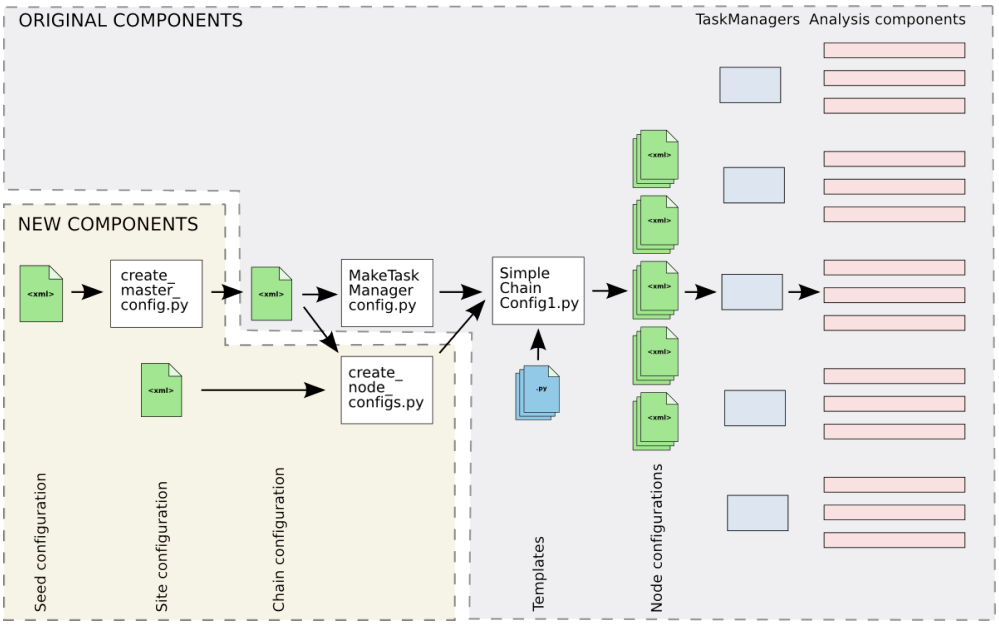

Figure 6.7.: Original and new components of the Configuration tools

functionality. Also, much work has been about refactoring and getting a framework for testing in place, and can therefore hardly be considered breaking new ground, although programs were created from an idea of how something should be rather than borrowing from known methods.

7. Evaluation and testing

7.1. Test methods

Unless anything else is explicitly stated, all tests that compare effectiveness of two or more pieces of code in a given application, are run on identical hardware. For the simpler tests the Unix tool time might be used instead of i.e. Python modules.

7.1.1. Regression testing

From the very beginning when rewriting XML parsing code, the need arose for some mechanism to see if changes made to the source code had had regressions. As there initially were no unit tests, it would be hard to make changes without introducing bugs, and to get in place unit tests would require refactoring to make the source code testable. To resolve this deadlock a program called test_tool.py was written.

While working with code it is important that functionality is not lost and that bugs are not introduced. The proof, so to speak, of a correct execution of the configuration tools is the output, that is the configuration and run-control scripts. For all new functionality added, the output of the new code should be checked against the previous version. The test_tool.py program takes as input directories from the site_config.xml file that define a directory of reference output and a directory for where set of chain configurations are stored. Then, for every chain configuration file, a run is made with the new code and the resulting files are compared to the reference output. The test script make use of these Python modules: shutil for high level file operations and filecmp for comparing files and directories.

7.1.2. Profiling - testing for effectiveness

As one of the objectives has been to increase execution speed, the use of tools to look where in the code the program is spending time can be useful for guiding optimization. Python has several built-in modules that can be used in this regard. hotspot was chosen due to the focus of minimizing overhead while profiling and rich documentation[54]. This module is used by adding a small amount of code to the source that is about to be tested.

7.2. Test results

This section presents the results from the tests that have been performed.

7.2.1. Rewrite of XML parsing code

Most crucial for testing at this stage was to ensure that the changes had not introduced any errors. Changing the XML parsing code could potentionally cause many unexpected changes to objects created by this part of the code. All configurations that was run with the two codebases were equal. That is, the new code had exactly the same output as the older code. There were several configurations that caused the execution to halt on error on both code bases, probably due to not being updated to be used for new versions of the configuration software.

Testing the effectiveness of the new code was done with the hotspot module mentioned above. The results show that the new code uses 6.485 seconds while the old code spends 12.479 seconds doing the same thing. Almost twice as fast. One indication of where this rather large speed increase can have its origins, is when looking at number of function calls. This number went down from 393062 to 238619 when changing to libxml2. This is also reflected in the function call listings, where functions are ordered by internal time (total time spent on this particular function) and call count. In the listings for profiling the original code, several functions from the PyXML module ranks high, while the functions from libxml2 figured rather low on the second profiling. (DO? add profiling results to appendix?)

XML Package	Execution Time	Function Calls
libxml2	6.485	238619
PyXML	12.479	393062

Table 7.1.: Performance XMLConfigReader.py libxml2 vs. PyXML.

7.2.2. Single Node Mode

For testing single node mode the Unix tool time has been used. Five machines made up the testing environment. Single sign-on was implemented with OpenSSH, MIT Kerberos and pam_krb5. That means that a user who uses ssh to log into one machine with username and password on the system, will have to supply a username and a password for the first machine, but for every other machine the user logs into, kerberos forwarding ticket system will automatically handle authentication without the user having to enter their credentials.

AFS and pam-afs-session is also included in the setup to provide a seamless integration of AFS distributed file system. Unfortunately, kerberos authentication of the AFS pam session has for unknown reasons been very slow and therefore timing the execution time has been done locally on the machine generating configs instead of timing the entire time for the ssh command to complete. i.e. ssh time MakeTaskManagerConfig.py, instead of time ssh MakeTaskManagerConfig.py. This was seen as an acceptable compromise as testing was possible and fixing the slow authentication of AFS was considered to be of lower priority.

Testing showed that single node mode was between five and six times as fast as normal mode for reasonably realistic configurations (see appendix A for more information).

7.2.3. NBus

NBus has only been an attempt to prototype an alternative solution and have therefore received very little testing. It was also decided that other things/ MakeTaskManagerConfig features should be prioritized ahead of NBus as being more important for the project. Development therefore stopped at the point that the framework was able to run two small test applications and a very simple analysis chain.

The test applications were a simple linux terminal and small tool for retrieving information about connected nodes. Both were implemented as gui applications with PyQt4. The terminal would let the user type in unix commands and use NBus to send them to all connected nodes. Response from the nodes would be collected by NBus and communicated as signals to the client application. The information application works in a similar way, exept using Python methods to gather information about the remote machine rather than the command line.

The simple test configuration that was brought to running state ran a single process on a single node.

7.3. Evaluation

In general, the modifications and improvements made to the source code have had the desired and expected effects. Although the source code at this point appears to be in a state of being composed of two different coding styles, it should hopefully emerge as clearer, shorter and more expressive once refactoring has been completed. The addition of unit tests and acceptance test should further improve the health of the code base.

The chosen technology has filled its intended role nicely. Performance has improved due to profiling and introduction of libxml2 as can be seen elsewhere in this chapter. Expanded usage of XML has helped create simpler tools for the users. Also for the NBus, the implementation came together as was intended with the components that was decided upon. All of Twisted, Avahi and D-Bus performed their tasks for which they were chosen as well as or better than expected.

In contrast, development methodology has not worked out entirely as well as hoped. The model that was initially drafted fell, to a certain degree, apart because of lack of communication. Due to the distance between customer and the developer, it was not possible at all to interact directly. Because the customer also was very busy during most of the development, it was hard to even keep up communication per mail or telephone. As a results, many decisions had to be made by the developer without the feedback that should ideally have been present.

In summary, compared to the targeted results, the outcome has for the most part been satisfactory. The possible exception being the chosen development methodology.

8. Conclusion

The project this thesis is involved with is a place where initiative can be taken and responsibility is given. It leaves the student with a lot of freedom with regards to implementation, which might seem overwhelming, but also is greatly appreciated. It is also satisfying to be able to possibly contribute to such an interesting system as the HLT is.

From the student's side, it was expected that development would contain a feedback cycle where small parts of software was written and submitted for comments. Then, either included in the HLT distribution or reworked until a an acceptable implementation was reached.

In hindsight it might not have been a realistic approach, as the available (people) resources that has the required knowledge about the configuration tools are limited. HLT software is also frequently used in tests and commissioning. One might therefore be reluctant to introduce new code in such a critical environment. A more independent approach where changes could be done more freely might have proven to be the better approach in order to complete the most tasks. Aspects of the implementation could then have been communicated on a more abstract level (rather than source code), which would have been less demanding to make work.

The stepwise methods from XP that was sought to be part of the learning experience, has therefore been less prominent than what was planned and rather a large merge of the desired features will have to take place at the end of the project.

To have made clear what could be expected could have made it easier to more quickly find a proper form of development. The student could certainly have put more pressure on the assigners of the tasks for providing feedback, but it is understood that the project is very busy at the moment and that one might be better off working independently from the initial assignment.

What should also be mentioned is that when working for a project like HLT, there is to a lesser extent than what might be usual, a beginning and an end. This work is part of a long string of development projects that goes on for a period of time that is way beyond the scope of a master thesis. This assignment has to be viewed in light of this. Most likely the work will continue on what has been developed and evolved with the overall system as it is put together. Therefore, forming a proper foundation might be more important than finishing all implementation tasks.

All things considered, this has been taken as an opportunity to experiment with alternative implementations and in the end it turned out to be a unique experience that has been greatly appreciated.

8.1. Summary

Working with this thesis has been both challenging and rewarding. It has required a lot of work and due to some of the choices made (i.e. using Gentoo, deploying one's own "cluster"), the student might have created a little bit more work than what was strictly necessary. It is, however, believed that these choices have brought important insights that adds to the experience with the thesis itself.

Although not all tasks have been finished, the student is satisfied with the results as most of the tasks where brought to a point where there was a working implementation. Most goals are therefore reached, but there will still be some work left to integrate the software with the main repository and if a refactoring is completed, some of the tasks will have to be re-implemented.

When looking back at how the project has unfolded, it is evident that a lot of experience has been gained. In many cases there are several things that would have been nice to know before the project started, so that certain problems could have been more easily tackled, but then again, one of the main purposes of such a thesis might be exactly to gain these kind of insights by experiencing them first-hand.

The things that in the end turn out to be important might be surprising, although having been taught in lectures as part of the curriculum in the master program. The more pertinent lessons learnt for this particular thesis can be summarized in these three points:

- **Refactoring** should be done first. It is a way for the developer to get the source to speak "their" language. There are some pre-conditions to refactoring, for instance unit tests. If these are missing or anything else is, that the developer needs to feel comfortable changing the source code, then add it. To be confident that the software operates correctly is of utmost importance to a successful implementation (to even make it possible to move forward).

- **Keep focused** on a small set of tasks at the time. Finish them and first then start on new tasks. There is often the temptation to start something new before current tasks are finished. The last bit is often the part that takes the longest time to finish and it can be hard to remember that it is not done before it is done.

- **Textbooks** teach methods and processes that can be applied in the development of software. Be it XP, UP or any other methodology. In practice these methods might be hard to properly exercise in an actual project. One will have to keep in mind what has been learned, but be aware that there is no easy answer to how software should be developed.

8.2. Further work

There might seem to be a heavy emphasis on refactoring in the later parts of this thesis, but this is only a testament to the lessons learned. The entire purpose of

refactoring is to make source code understandable. Although the original sources of the configuration tools are not that large (approximately 4000 lines, 7000 including templates and all), it was nevertheless hard to get a good grasp of how the system worked.

If there was more time, a proper refactoring would for sure have been the highest priority, then a re-implementation of the tasks that needed it and integration with the main repository.

The only two task that has no implementation whatsoever, is "not recompile byte-code" and "repeated creation of configuration elements". It is not entirely clear if using the NBus approach would solve these tasks completely, as the implementation was not taken long enough to give a definitive answer.

A. Single-node mode test

A.1. Test setup

A.1.1. Nodes

The capabilities of the nodes are listed below in the table A.1

Node	Cpu	MHz	Bogomips	Memory
fuzzy	AMD-K7(tm) Processor	550.996	1103.13	515964 kB
kallen	AMD Athlon(TM) XP 2400+	2010.306	4023.56	514632 kB
katten	AMD Athlon(tm)	1202.760	2407.35	515988 kB
knutn	AMD Athlon(tm) XP 2500+	1852.085	3707.34	1034640 kB

Table A.1.: Machine properties.

A.1.2. Tools

All timings are found by using the unix command, time. All configurations are created from seed config files with the create_master_config.py script. Then create_nodes_config.py is used for single node mode and create_nodes_config.sh is used for normal mode. Site specific configuration, such as paths and more, are placed in site_config.xml.

A.1.3. Infrastructure

The setup uses kerberos + ssh + pam_krb5 for single-sign-on logon and AFS for distributed filesystem.

A.1.4. Notes

Recorded time measured in time it took to run the process on the node, excluding ssh. Ssh being strangely slow at the time the tests was executed. First run always slower, probably due to files being created, then later only opened.

A.2. Results

The results are average over 10 runs in a while loop in a bash script. To make the two test cases comparable, normal mode is always executed on the node "knut" and the heaviest load for single node mode is also placed on "knut". The different configurations are listed below and results can be found in table A.2:

1. One sector. The test setup consists of four nodes. The configuration file describes three nodes running two FilePublishers and two ClusterFinders each, and one node running a SliceTracker and a TCPDumpSubscriber. Normal mode is executed on the node named "knut" and single node mode ran with heaviest load on the node "knut". The two test cases should therefore be comparable.

2. One side. All processes on available nodes. The configuration for one sector replicated over the 18 sectors a complete side consists of. The configuration describes a setup where all tasks are run on the four available nodes.

3. Executale (almost) configuration. Three nodes running a FilePublisher and a ClusterFinder each. One node running SliceTracker and TCPDumpSubscriber.

Mode/Configuration	1	2	3
Normal	2.10 sec	1 min 2 sec	1.0 sec
Single node	0.36 sec	33.5 sec	0.2 sec
Ratio normal to single node.	6	2	5

Table A.2.: Results single node test.

A.2.1. Conclusion

The actual configuration running on the hlt cluster should be more similar in nature to the test with one sector. The node running the create_node_configs.py script would have to iterate over all participating nodes and send commands over ssh/rsh, but the rest of the nodes will only do work concerning their own configuration. The execution of commands over ssh/rsh can be backgrounded so that a command will not block the following command while it finishes its execution. Significant runtime savings should therefore be expected by distributing the task of configuring the nodes to the nodes themself. The results show that single node mode is between two and six times faster, depending on configuration.

A.2.2. Comments

In this setup, the master configuration file is shared over an openafs filesystem, so that it is accessible from all nodes. But, the node configuration files (and shell scripts) generated at each node is saved locally, thus using faster data storage and reducing

network traffic caused by saving over the distributed filesystem. This organization does not inflict upon the operation of the TaskManager. I.e. when starting a chain with the runscript, it will find the correct paths to config files and executables.

B. Sample master configuration file

Listing B.1: Example of configuration for one patch

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SimpleChainConfig1 ID="pp_test" verbosity="0x39">

<!-- ========================================================== -->
<!-- ====================  NODELIST  ========================== -->
<!-- ========================================================== -->
<Node ID="master" hostname="knutn" />

<!-- ########################################################## -->
<!-- ####################  Patch 0  ########################### -->
<!-- ########################################################## -->
<Proc ID="FP_0_0" type="src">
    <Cmd>FilePublisher
        -datafile /data/raw0/TPC_22.ddl
        -datatype DDL_RWPK -dataspec 0x0B0B0000 -dataorigin
            TPC
        -eventtime 200000 -sleeptillevent
    </Cmd>
    <Node>master</Node>
    <Shm blocksize="40k" blockcount="12" type="sysv" />
</Proc>

<Proc ID="CF_0_0" type="prc">
    <Cmd>AliRootWrapperSubscriber
        -componentid TPCClusterFinderPacked
        -componentlibrary libAliHLTTPC.so
        -componentargs "pp-run rawreadermode offline"
    </Cmd>
    <Parent>FP_0_0</Parent>
    <Node>master</Node>
    <Shm blocksize="125k" blockcount="12" type="sysv" />
</Proc>

<Proc ID="TDS" type="snk">
    <Cmd>TCPDumpSubscriber -port 42000</Cmd>
    <Parent>FP_0_0</Parent>
```

```
36      <Parent>CF_0_0</Parent>
37      <Node>master</Node>
38      <Shm blocksize="4k" blockcount="1" type="sysv"/>
39  </Proc>
40
41  </SimpleChainConfig1>
```

Glossary

AFS Andrew File System. 34, 70, 75

Alice A Large Ion Collider Experiment. 2, 6–9, 11, 32

AliRoot ALICE Off-line framework. 18, 43, 45

API Application programmer interface. 53

ATLAS General-purpose experiment that is part of LHC. 7

BIOS Basic Input/Output System. 15

CERN European Organization for Nuclear Research. 2, 6, 7, 11, 18, 31, 45

CHARM Remote management card used in the HLT cluster. CHARM is an acronym for Computer Health Analyser and Remote Management[55]. 15, 49

CLI Command Line Interface. 51

CMS General-purpose experiment that is part of LHC. 7

COTS Common-Off-The-Shelf. 9, 15

CPU Central Processing Unit. 9, 15, 49

D-RORC DAQ - Read-Out Receiver Card. 13

DAQ Data Aquisition. 8, 9, 13, 15

DDL Detector Data Link. 8, 9, 26

DOM Document Object Model. 34, 46, 47, 55, 58

FEP Front End Processor. 9, 13, 15, 26

FOSS Free and Open-Source Software. 49

FPGA Field-programmable gate array. 15

GUI Graphical user interface. 22, 23, 35, 51

H-RORC HLT - Read-Out Receiver Card. 13, 15

HLT High-Level Trigger. 2, 6, 8, 9, 11, 13, 15–18, 20, 21, 23, 25–27, 30–32, 34, 42, 43, 45, 49, 58, 72

HMPID High-Momentum Particle Identification Detector. 8

ITS Inner Tracking System. 8

LEP Large Electron-Positron Collider. 7, 8

LHC Large Hadron Collider. 6, 7, 31

LHCb Experiment that will study B-hadrons and CP violation at LHC.. 7

LHCf Experiment that will perform measurement of forward neutral particle production for cosmic ray research. 7

OS Operating System. 44

PCI Peripherial Component Interconnect. 15

Perspective Broker A translucent reference to a remote object[49]. 52

phos Photon Spectrometer. 8

ROOT An Object-Oriented Data Analysis Framework. 18, 45

RPC Remote procedure call. 50, 65

SAX Simple API for XML. 34, 46, 47, 55

SCM Source Code Management. 43

SGML Standard Generalized Markup Language. 46

SLC Scientific Linux CERN[18]. 31, 49

SMP Symmetric Multiprosessing. 18

Standard Model Grouping of two major theories, quantum electroweak and quantum chromodynamics, which provides an internally consistent theory describing interactions between all experimentally observed particles[5]. 6

TOF Time-Of-Flight. 8

TOTEM An experiment dedicated to the measurement of total cross section, elastic scattering and diffractive processes at the LHC.. 7

TPC Time Projection Chamber. 8, 9, 13, 15, 18, 23, 27

TRD Transition-Radiation Detector. 8

UML Unified Modeling Language[6]. 44, 48

UP Unified Process. 39, 73

VNC Virtual Network Computing[5]. 15

XML Extensible Markup Language. 11, 18, 23, 27, 33–37, 44, 58, 69, 70

XP Extreme Programming. 29, 32, 39–41, 72, 73

XPath XML Path Language. 34, 46, 47, 55

Bibliography

[1] Cern document server - http://cdsweb.cern.ch/, August 2007.

[2] Orlando Villalobos Baillie, Pierre Van de Vyvre, D Rorich, V Lindestruth, Lennart Jirdén, Hans de Groot, Christian Wolfgang Fabjan, and Lodovico Riccati. *ALICE trigger data-acquisition high-level trigger and control systemTechnical Design Report*. Technical Design Report ALICE. CERN, Geneva, 2004.

[3] Timm Morten Steinbeck. *A Modular and Fault-Tolerant Data Transport Framework*. PhD thesis, Universität Heidelberg, Fakultät für Mathematik und Informatik, 2004.

[4] Timm M. Steinbeck, Volker Lindenstruth, and Heinz Tilsner. A control software for the alice high level trigger. Technical report, Kirchhoff Institute of Physics, Ruprecht-Karls-University Heidelberg, Germany, for the ALICE Collaboration, 2004.

[5] Wikipedia - http://www.wikipedia.org/, August 2007.

[6] Object Management Group. *Unified Modeling Language (UML), Version 2.1.1*. OMG, http://www.omg.com/uml/, 2007.

[7] Cern public webpage, about cern, http://public.web.cern.ch, August 2007.

[8] Infiniband - http://www.infinibandta.org/home, October 2007.

[9] Root - http://root.cern.ch/, October 2007.

[10] Geant- http://wwwasd.web.cern.ch/wwwasd/geant/, October 2007.

[11] Aliroot - http://aliceinfo.cern.ch/offline/, October 2007.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[13] Pyqt - http://www.riverbankcomputing.co.uk/pyqt/, August 2007.

[14] Qt - http://trolltech.com/products/qt, August 2007.

[15] Torstein Thingnæs. Generering av konfigurasjonsfiler for taskmanager i hlt-systemet for alice-eksperimentet på cern. Master's thesis, Universitetet i Bergen, June 2007.

[16] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[17] Ian Sommerville. *Software Engineering*. Addison Wesley, 2001.

[18] Scientific linux cern - http://linux.web.cern.ch/linux/scientific4/, sep 2007.

[19] Ubuntu - http://www.ubuntu.com/, September 2007.

[20] Craig Larman and Victor R. Basili. Iterative and incremental development: a brief history. *IEEE Computer*, pages 47–56, July 2003.

[21] Ken Schwaber. *Agile Project Management With Scrum*. Microsoft Press, Redmond, WA, USA, 2004.

[22] Java - http://java.sun.com/, October 2007. http://java.sun.com/.

[23] Python - http://www.python.org/, August 2007.

[24] Ironpython - http://www.codeplex.com/ironpython, October 2007.

[25] Jython - http://www.jython.org/, October 20007. http://www.jython.org/.

[26] Bourne-again shell - http://tiswww.case.edu/php/chet/bash/bashtop.html.

[27] Tim Bray, Eve Maler, François Yergeau, C. M. Sperberg-McQueen, and Jean Paoli. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[28] Sax - http://www.saxproject.org/, October 2007.

[29] Dom - http://www.w3.org/dom/, October 2007.

[30] James Clark and Steven DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[31] Elementtree - http://effbot.org/zone/element-index.htm, October 2007.

[32] Steven DeRose and James Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[33] Martin Fowler. *UML Distilled*. Addison Wesley, 1997.

[34] Linux - http://www.kernel.org/, October 2007.

[35] Gentoo - http://www.gentoo.org/, October 2007.

[36] Openssh - http://www.openssh.com/, October 2007.

[37] Nomachine - http://www.nomachine.com/, October 2007.

[38] Kde - http://www.kde.org/, October 2007.

[39] Kate - http://kate-editor.org/, October 2007.

[40] Konsole terminal - http://konsole.kde.org/, October 2007.

[41] Diffutils - http://www.gnu.org/software/diffutils/, October 2007.

[42] Subversion - http://subversion.tigris.org/, October 2007.

[43] Kile - http://kile.sourceforge.net/, October 2007.

[44] Uml - http://uml.sourceforge.net/index.php, October 2007.

[45] Kdiff3 - http://kdiff3.sourceforge.net/, October 2007.

[46] Kompare - http://www.caffeinated.me.uk/kompare/, October 2007.

[47] D-bus homepage - http://www.freedesktop.org/wiki/software/dbus, September 2007.

[48] Avahi homepage - http://avahi.org/, September 2007.

[49] Twisted homepage - http://twistedmatrix.com/trac/, September 2007.

[50] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[51] Martin Fowler. Continuous integration, May 2006.

[52] Portage - http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1, October 2007.

[53] Pyxml - http://pyxml.sourceforge.net/, September 2007.

[54] Python documentation - http://docs.python.org/, September 2007.

[55] Hlt wiki - http://wiki.kip.uni-heidelberg.de/ti/hlt/index.php/main_page, September 2007.