Phronesis, a diagnosis and recovery tool for system administrators

Christophe Haen

N° d'ordre : D. U : 2388
E D S P I C : 620

UNIVERSITÉ BLAISE PASCAL - CLERMONT II

ECOLE DOCTORALE
SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

Thèse

Présentée par

Christophe Haen

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

SPECIALITE : INFORMATIQUE

Phronesis, a diagnosis and recovery tool for system administrators

Soutenue publiquement le 24 octobre 2013 devant le jury :

M. David Hill	Président
M. Abdel-Illah MOUADDIB	Rapporteur et examinateur
M. Philippe PREUX	Rapporteur et examinateur
M. Niko NEUFELD	Examinateur et superviseur
M. Vincent BARRA	Directeur de thèse

Acknowledgments

First of all, I would like to express my deepest gratitude to Prof. Vincent Barra, my thesis advisor, for agreeing on an extremely short notice to supervise me for this project at CERN. Throughout the three years that my work lasted, he has always and at all times provided me with all the help I was in need of, despite the distance. I have greatly benefited from his knowledge of the various artificial intelligence domains, his experience of the research world, and discussions with him were always interesting, enjoyable and fruitful. Without his guidance and persistent help this thesis would not have been possible.

Many thanks to my committee, Prof. David Hill, Prof. Abdel-Illah Mouaddib and Prof. Philippe Preux for their guidance and helpful suggestions.

Dr. Niko Neufeld, deputy of the LHCb Online project, also deserves a special thanks. As my official CERN supervisor, he has given me many opportunities to get involved in several aspects of the Online work and the data acquisition world. All the experience gathered there had a direct impact on my way of working and made me professionally proficient.

I am also indebted to each and every person in the LHCb Online team: Enrico Bonaccorsi (grazie!), Loic Brarda (merci!), Daniel Campora(¡gracias), Luis Granado Cardoso (obrigado!), Mohamed Chebbi (merci!), Paolo Durante (grazie!), Markus Frank (danke!), Clara Gaspar (obrigado!), Beat Jost (danke!), Vijay Kartik (sorry dude, no latex support for you ;-)), Guoming Liu (XiéXié!), Francesco Sborzacchi (grazie!), Rainer Schwemmer (danke!) and Alexander Zvyagin (spasibo!). They taught me all what I know about system and network administration and the running of the LHCb experiment. They also made my three years at CERN a unique and very pleasant human adventure.

Needless to say that there are many other people at CERN who had a positive impact on my work one way or another. However, I will not take the risk of making a list for fear of forgetting any, but it definitely includes the wonderful LHCb secretariat (to whom I am particularly grateful for protecting me from all kind of administrative hassles).

Finally, thanks to AC↯DC, Eric Clapton, Deep Purple, B.B. King, Mark Knopfler, the Rolling Stones, Sigur Ros and ZZ Top to cut myself off from the surrounding noise when writing this thesis. They all have a drink on me.

Abstract

The administration of a large computer infrastructure is a great challenge in many aspects and requires experts in various domains to be successful. One criterion to which the users of a data center are directly exposed is the availability of the infrastructure. A high availability comes at the cost of constant and performant monitoring solutions as well as experts ready to diagnose and solve the problems. It is unfortunately not always possible to have an expert team constantly on site.

This work presents a tool which is meant to support system administrators in their tasks by diagnosing problems, offering recovery solutions, and acting as a history and knowledge database.

We will first detail what large data centers are composed of and what are the various competences that are required in order to successfully administrate them. This will lead us to consider the problems that are traditionally encountered by the administrators. Those problems are at the source of this project, and we will define our goals from those issues. Finally we will detail the environment in which this work took place, that is the LHCb experiment at CERN.

The second chapter contains the state of the art, which lists the methods that try, one way or the other, to answer problems similar to ours. We will see what these methods are, how they are applied, what the pros and cons are, and how they fit in our context.

The third chapter is our proposition to address the issues and reach the goals we set in the first chapter. This chapter gives all the approaches and methods we adopted: some are a direct use or an adaptation of what was mentioned in the state of the art, while some others are innovative techniques. Each proposition is justified and put in relation with our goals.

The fourth chapter is a technical description on how our ideas of the third chapter were put in practice. It details the software packaging, the various tools that were developed as well as some explanations on how they are used.

The fifth chapter shows what results were obtained with your solution. The first part of it is an analysis of various simulations that aim at showing the efficiency of the methods described in Chapter 3. The second part is a description of how the software is applied in the LHCb Online environment, as well as a feedback on its usage.

The last chapter offers perspectives to improve the software, extend its functionalities and address some of the issues that are detailed in the feedback of Chapter 5.

Contents

List of Figures

List of Tables

Chapter I

Introduction

Contents

I.1 Administration of a large computer fabric

A data center or computer center is a facility used to house computer systems and associated components, and are usually meant to fulfill a particular goal. Typical goals include data storage, website hosting, cloud facilities or heavy computation. The size and the components of a computer center strongly depend on the usage, but some elements at the core of the facility are always present:

- Servers: these are of course the basic block of a data center and normally represent the greatest share of the devices.
- Network components: the network infrastructure interconnects all the servers, and is composed of various equipments such as switches, routers, patch panels and of course a lot of cables.
- Storage: the storage system is usually a large pool of disks used to centrally host the data and share it across the data center.
- Infrastructure equipments: cooling, power, UPS, racks... as many components on which the computer infrastructure is based and relies.

Some middle-wares are not mandatory but are very frequent, such as:

- Databases: a specific setup optimized for the use of databases
- Virtualization: more and more common now, virtualization is a technique for using computing resources and devices in a completely functional manner regardless of their physical layout or location.

Taking care of such a data center implies a wide range of competences and different tasks:

- Hardware: be it server or network hardware, choosing suitable equipment is crucial. Considerations such as performance, power consumption, or compatibility are only a few of the aspects to look at when choosing material, and one always has to consider them all at the same time. For example, a server might be more expensive than two servers twice as less powerful, but be more adapted regarding power consumption or space occupation. The *"power usage effectiveness"* (PUE) which measures how efficiently a data center uses its power is also taken more and more into account now. Also, one should not forget that each hardware model has not only different specifications, but also different problems and malfunctions. And thus the more heterogeneous an environment is, the more varied problems one might have to face. Proper cabling becomes critical very quickly with the number of devices to interconnect: one has to choose adapted lengths, make clever connection designs, and so on.

- Networking: having all the hardware neatly interconnected is a first step, but the logical interconnection is the next one. Networking experts have to divide the infrastructure into logical sets of machines either to separate the different functionalities of the center or to allow for more performances. But they have to ensure as well that the parts that need to communicate can actually do it. All this is achieved by designing clever addressing scheme and making use of various protocols of routing and segmentation.

- Storage: the storage is the system on which shared data are kept. It can easily become a single point of failure. Just as for the hardware and the networking, the so-called *"good"* storage system depends on the use case. Some systems like tapes are targeted at providing huge storage capacity in the long term, typically for backups, which they achieve at the cost of performance. Other systems provide very fast access to data, but usually offer limited storage capacities. The capacity and the speed are not the only aspects to consider when designing the storage system. Fault tolerance, recovery capacities, cluster or single access, are to be taken into account. Storage systems nowadays always rely on a RAID-like system, which involves aggregating disks in various manners in order to overcome the limitations of the hardware disks [Patterson 1988]. For example, RAID 0 improves the speed by striping the data and writing the various chunks simultaneously on several disks; RAID 1 provides better reliability because it duplicates all the data, at the cost of losing half the storage capacity; more modern versions like RAID 5 or 6 improve both the speed and safety by striping the data and using checksums for integrity.

- Software: In order to be used for anything, the data center has to run software solutions, the first of them being the operating system (OS). The choice of the OS is usually driven by the activity the data center is dedicated to, or sometimes enforced by external constraints, and it is not rare to encounter

several OS's or even several versions of the same OS in an environment. But it's/their deployment, and of any other application on top of them cannot be approached in the same way as on a desktop computer (i.e. with an installation CD). It requires special tools called *"cluster management systems"* (CMS), which allow to deploy software and configurations at a large scale. Of course, each OS has a different cluster management tool, thus duplicating the maintenance work for the administrator.

- Users: some data centers do not give direct access to their machines to the users, but when they do, there is the need for a whole infrastructure in order to manage the user profiles. This infrastructure needs to cover all the aspects regarding the accounts, the authentication, the authorizations and so on. As for the CMS, a unified system covering all the OS is not a trivial goal.

- Security: this is a vast subject, which unfortunately can never be completely solved in an environment. The strength of a chain is in its weakest link, the same applies for security. And the security of a data center covers many layers: network, software, users and more. Security experts have to decide what they want to protect, against which type of threat, and how.

This list is far from being exhaustive, but already shows the complexity of running a data center. Moreover, the difficulty does not increase linearly with the size of the data center, but rather in a step manner. The reason is that all solutions have a threshold after which they cannot cope anymore; neither can we decide to put in place the most advanced solutions for low-end data centers because this would be an overkill and possibly out of budget.

The International Data Corporation (IDC) [Vernon Turner 2005] has classified (in 2006) the data centers into four categories, distinguished by size and based on three criteria: the floor surface, the total amount of servers, and the amount of high end servers. This classification is now deprecated, given the speed at which the field evolves, but still of interest:

- Small Data Center:
  - 1400 m^2
  - 350 - 500 volume servers
  - 1 - 3 high end servers
- Medium Data Center:
  - 1850 m^2
  - 1500 - 1700 volume servers
  - 4 - 5 high end servers
- Large Data Center:

  - 3250 m^2
  - 2000 - 2500 volume servers
  - 6 - 7 high end servers

- Very Large Data Center:
  - >9300 m^2
  - <25000 volume servers
  - >8 high end servers

We shall now mention the problems that can be encountered when managing a data center.

I.2 Problems

In the previous section, we have listed only some of the competences needed to successfully run a data center, and mentioned only a fraction of the tasks it involves. An important aspect has nevertheless been moved aside: the availability. The expectations on the availability strongly depends on the usage of the data center. A commercial center will achieve a better availability than the cluster of a scientific public institute, because high availability has a non negligible cost: it is usually achieved by using higher quality material, having a fully redundant infrastructure, and having available spares for every component. The Telecommunications Industry Association (TIA) published and amended in 2010 a document TIA-942 [TIA 2010] dividing data center into tiers according to their availability. The higher the tier the higher the availability:

- Tier 1
  - Single non-redundant distribution path serving the IT equipment
  - Non-redundant capacity components
  - Basic site infrastructure with expected availability of 99.671%
- Tier 2
  - Meets or exceeds all Tier 1 requirements
  - Redundant site infrastructure capacity components with expected availability of 99.741%
- Tier 3
  - Meets or exceeds all Tier 1 and Tier 2 requirements
  - Multiple independent distribution paths serving the IT equipment
  - All IT equipment must be dual-powered and fully compatible with the topology of a site's architecture

  - Concurrently maintainable site infrastructure with expected availability of 99.982%

- Tier 4
  - Meets or exceeds all Tier 1, Tier 2 and Tier 3 requirements
  - All cooling equipment is independently dual-powered, including chillers and heating, ventilating and air-conditioning (HVAC) systems
  - Fault-tolerant site infrastructure with electrical power storage and distribution facilities with expected availability of 99.995%

While the unavailability tolerated for a tier 1 is almost 29 hours over one year, the one for a Tier 4 represents only 26.28 minutes.

So in fact, the main challenge of the administrator team taking care of the center is not to deploy it, but to maintain it in a working shape. In order to achieve their goal, the administrators need to react quickly to a failure to first diagnose the problem, and then solve it. Reaching the availability targets listed above, even for Tier 1, requires very fast intervention 24h / 7 days and implies having a working force on site at any time. This is a problem in most of the non-profit data centers, which usually have an *"expert on-call"* system: this means that in case of problem, at any time including the night, an expert will be contacted and eventually woken up to solve the issue. Properly addressing the problem requires taking the measure of the global situation, understanding what is impacted, recalling the interactions between all the different components of the infrastructure, tracing down or inferring the possible sources of problem, validating the diagnosis and finally taking actions. All those steps stand in need of a clear mind, which is not an easy thing when just having been roused. It usually results in a poor efficiency at the beginning of the intervention, and thus in a loss of time and availability of the data center.

Of course, the difficulty of keeping the data center in an appropriate shape is not only true in the middle of the night. However, this is the moment when it is the most difficult. The reason is that one does not necessarily have a clear mind, and because the expert might simply not know the whole system fully in details and is usually not surrounded by his team at this time of the day.

I.3 Goal

Efficiency is a parameter of the data center to which the users are directly exposed and very sensitive. So in cases where having a permanent on-site team of administrator is not an option, the straightforward alternative is to use a software solution. Software solutions might have an initial cost of development or acquisition, and eventually a certain running cost for licenses or support purposes, but on a long term scale, the price is usually negligible compared to the one of an employee.

Replacing a human administrator by a software is certainly not a reachable goal, at least for now, because of the variety and complexity of the tasks. But using a software as a support to the person is a more reasonable, common and fruitful approach.

In this particular context, where the goal is to support a system administrator to keep the data center running, and fix a problem when it occurs, there are characteristics and expectations that immediately come to mind regarding the software:

- Reduce the workload of the system administrator: although this statement can sound like an obvious requirement, one still has to bear in mind this constraint all along the development of the tool. Indeed, it is very easy to forget, and this generally leads to a software which is so complex, requires so much configuration and so much maintenance that using it is more time consuming that actually performing the task. So ideally what one wants is a tool which is as independent as possible.

- Propose a diagnostic and recovery solution: ultimately, the goal is a software which is able to tell where a problem comes from and how to fix it. This is definitely the best help a just woken up administrator can get.

- Improve with experience: we target at a tool providing more and more accurate results as it encounters situations. The same way as one can expect from a student that he uses what he has learned or what he has been taught, one can expect a similar behavior from this software.

- Act as a knowledge base: since the aim of the tool is to assist an administrator in maintaining a data center in a running condition, one can expect the tool to have some kind of knowledge about that environment: mainly, what are the different components, their behaviors, their interactions, and their dependencies. An experienced administrator starting a new job, if provided with all this knowledge, would be immediately efficient in its tasks.

- Act as a problem history base: problems are hardly unique. Experience shows that if a setup does not change, problems are recurrent. Their frequency might be very low, but they will eventually appear again. This *"fatality"* of problems is not true only across time, but also across similar systems: if several systems have a very similar design, then the chances for a problem to appear only on one of them is almost null. This is the reason why the software may have to remember previously encountered problems, and keep the history at the disposal of the administrator or the helpdesk.

The following addresses all these points, through the development of both a methodological framework and a dedicated tool for the LHCb Online team.

I.4 Context

CERN, the *"European Organization for Nuclear Research"*, sits on the Franco-Swiss border, next to Geneva. It is one of the largest and most prestigious scientific research centers across the globe. Founded in 1954, it is one of the first European organizations, and counts 20 Member States and 7 observers today. Its primary interest is fundamental physics: understanding the laws and constituents of our universe. Currently, there are 2,400 full-time employees, 1,500 part-time employees, and some 10,000 visiting scientists and engineers, representing 608 universities and research facilities and 113 nationalities, that are dedicated to this task. CERN has a great number of achievements in its fundamental research. One can mention:

- discovery of neutral currents
- discovery of W and Z bosons
- determination of the number of light neutrino families
- first creation and longest time maintaining antihydrogen atoms
- discovery of a Higgs-like boson

Several Nobel prizes and many other valuable titles and medals have been awarded to CERN scientists.

In order to reach such achievements, physicists need tools and materials that are at the bleeding edge of the technologies, and sometimes even beyond immediate reach. Those needs explain the multitude of competencies at CERN: the goal of the biggest fraction of CERN scientists and engineers is not to develop new fundamental theories but to develop the technologies and operate the advanced tools in order to provide the physicists with data. This mission has brought many technological breakthroughs, which are nowadays used daily in a variety of fields: medical imagery instruments, solar panels, cooling technologies, large scale computing, and, last but not least, the Web.

Across its history, CERN has always operated particle physics instruments that were at the time setting new records in terms of size, power, temperature or other values. In 2008, CERN has overcome more limits than ever before with the LHC.

The LHC, Large Hadron Collider, is the latest particle accelerator at CERN. Particle accelerators are at the core of high energy physics (HEP). Their goal is to accelerate particles at very high speed and energy in order to observe events that are not visible at energies we encounter every day. The LHC is the most recent particle accelerator, and is the most complex machine human beings have ever built. Accelerators are either linear or circular. While linear accelerators were the first to be used and will be the next generation, the current generation is mainly

circular accelerators, and so is the LHC.

Very roughly described, a circular accelerator is made of two beam pipes in which one accelerates particles in opposite directions via a radio-frequency system. The particles accelerated might either be of the same charge sign like proton-proton, proton-ions or ions-proton for the LHC, or of opposite charge, for example electron-positron. Those beam pipes are inside cylindrical magnets (dipoles in Fig. I.1, quadrupoles, and even octupoles) in order to bend particles' trajectory or shape the particle clouds, which explain the size of these instruments. The pipes are then crossed at definite places, bringing the particles into collision. The reason for such an operation is to understand the building blocks of our universe by creating particles that are too heavy to be present naturally in our environment: collisions result in high energy generation, which in turn can transform into matter according to Einstein's special relativity theory.

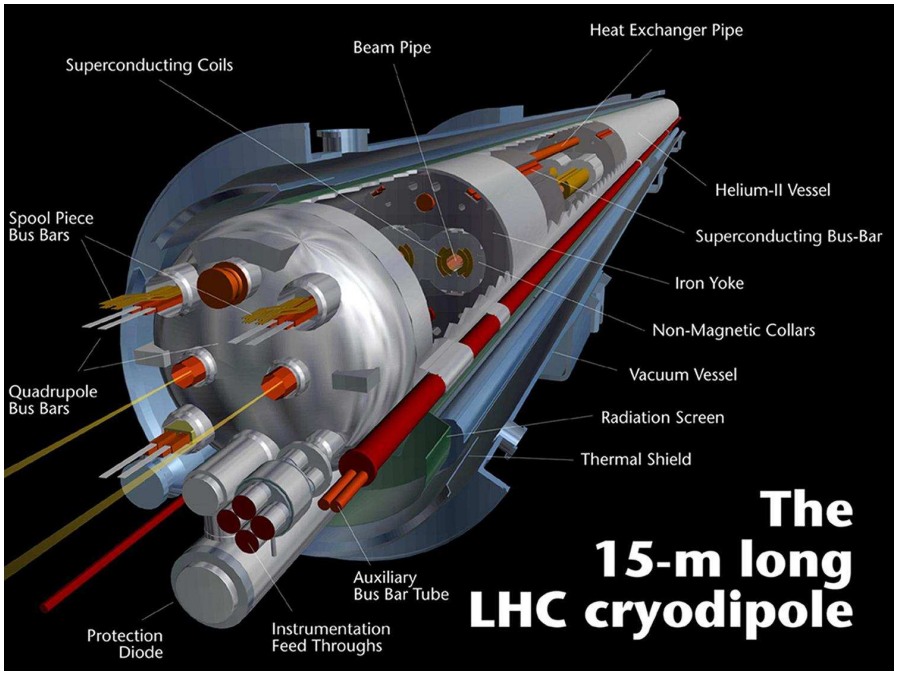


Figure I.1: An LHC dipole

The LHC sits 100 m below the ground, under the Franco-Swiss border, in a 27 km circumference tunnel as visible in Fig. I.2. This machine has outdone any previous comparable instrument in all regards:

- With a temperature of about 2 K, the 1600 superconducting magnets of the LHC are among the coldest places of the universe, chilled with almost 100 tons of liquid Helium.

- The field of the dipole magnet goes up to 8.3 T, compared to the Earth field which is of 31 μT.
- The vacuum present in the beam pipe is emptier than inter-galactic space.
- The particles travel at a speed of 0.999999991 c, which is only 3 m per second slower than the speed of light.
- Each proton will have an energy of 7 TeV.
- A luminosity of 10^{34} $cm^{-2}s^{-1}$.
- A collision rate of 40 MHz, or a bunch collision every 25 nanoseconds.

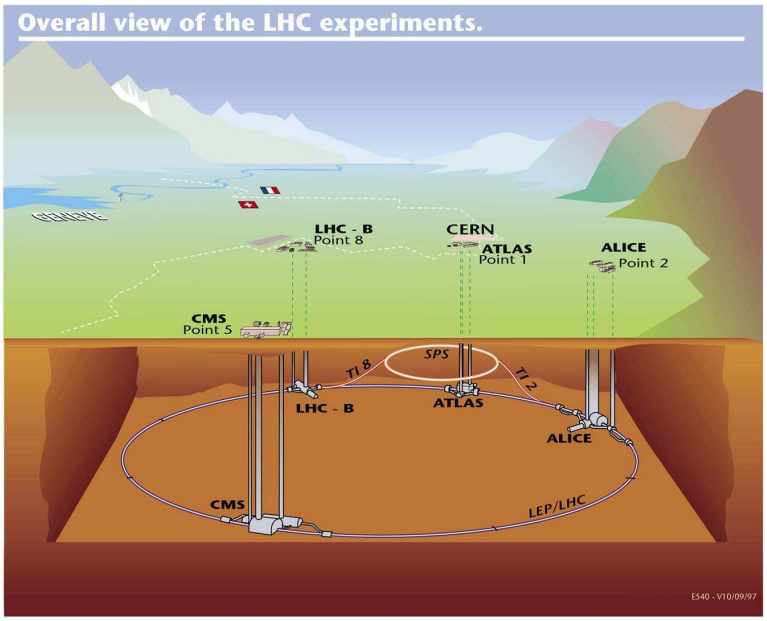


Figure I.2: Overview of all the LHC experiments

However, colliding particles is pointless if there is nothing to observe the result. This is the purpose of the different detectors which sit on top of each of the four interaction points of the LHC. Detectors are like big cameras that take snapshots of what is happening when the collisions take place. There are four main experiments using the collisions of the LHC:

- ALICE: *"A Large Ion Collider Experiment"*. The aim of this experiment is to observe the Quark Gluon Plasma, a state of matter that is supposed to have existed just after the Big Bang. In order to recreate this state, the LHC will collide lead's ions, producing temperatures more than 100,000 times hotter

than at the heart of the Sun. Its dimensions are 26m long, 16m high and 16m wide and it weighs 10,000 tons.

- ATLAS: *"A Toroidal LHC Apparatus"*. It is one of the two giant multi purposes detectors of the LHC. Among its goals are the discovery of the Higg's boson, extra dimensions and dark matter. Its dimensions (46 m long, 25 m high and 25 m wide and 7000 tonnes) make it the largest volume particle detector ever built.

- CMS: *"Compact Muon Solenoid"*. CMS is the second multi-purpose detector of the LHC. It has the same physical goals as ATLAS, but uses different technical solutions. With 21 m long, 15 m wide, 15 m high and 12 500 tons, it is smaller but heavier than ATLAS.

- LHCb: *"Large Hadron Collider Beauty"*. The role of LHCb is to understand the difference between matter and anti-matter, and why all the latest seems to have completely disappeared from our Universe. For that, it observes particles of the *"charm"* and *"beauty"* families.

The standard theory used to describe particles physics is known as *"Standard model"*. It describes all the elementary particles, forces and their interactions. This model is extremely successful, but still suffers from certain shortcomings.

Particle physics theory has foreseen and confirmed the existence of the so-called *"antiparticles"*. Each *"standard"* particle has an antiparticle which has the same mass and spin, but opposite charge. When put together, a particle and its antiparticle annihilate, leaving only pure energy. The theory states that at the beginning of the Universe, particles and antiparticles were created in equal quantities. However, all that surrounds us is composed of ordinary matter, and there is no sign of evidence for an *"antiparticle reserve"* in the universe. This means that at some point in time, the balance was broken. The Standard Model predicts a small asymmetry, but not big enough to explain the quantity of matter we observe. The main goal of LHCb is to investigate this.

LHCb is different from the other LHC detectors by its geometry: while the others are cylinders, LHCb is a forward detector, as visible in Fig. I.3. With 21m long, 10m high,13m wide and 5600 tons, it is also the lightest and smallest of all four.

LHCb, like any other experiment, is actually made of several components, called sub-detectors, used together. One sub-detector is dedicated to perform one type of measurement. LHCb has the following sub-detectors, in the order of particle crossing:

- VELO: the VErtex LOcator is the first detector that the particles go through. It is a cylindric silicon detector around the interaction point, very close to the

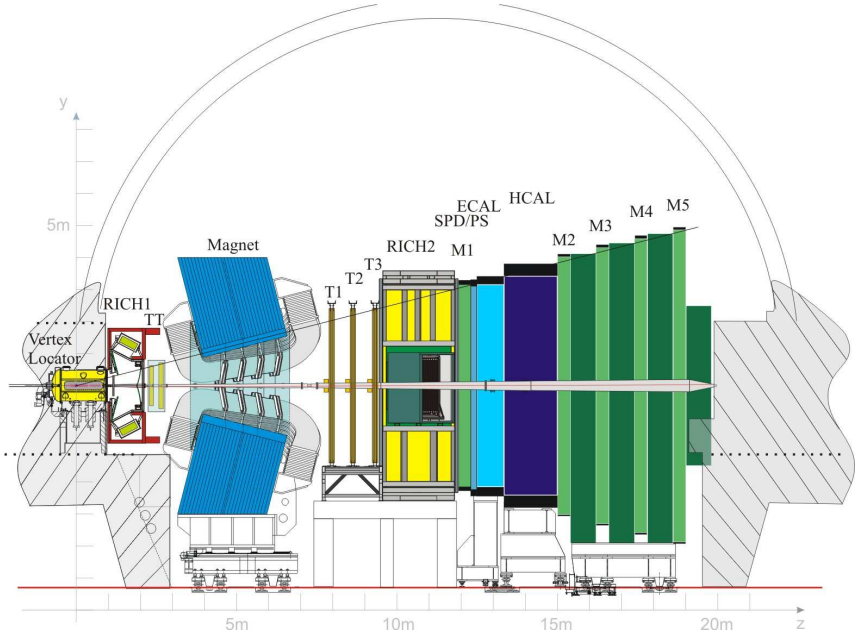


Figure I.3: Section of the LHCb experiment

beam pipe. It is used to measure the trajectories of particles right after the collision with extreme precision.

- RICH1: the Ring Imaging CHerenkov detector identifies the particles' with low momentum using the Cherenkov effect.
- TT: the Tracker Turicensis is a silicon strip detector used to measure particle's trajectories and their momenta.
- Magnet: the magnet is not a measurement instrument. It is used to bend particle's trajectories.
- IT and OT: three slices of a tracking system are placed behind the magnet. The inner part of it is the IT (Inner Tracker), a silicon strip detector; The outer part is the OT (Outer Tracker), a straw-tube detector.
- RICH2: it is used as RICH1 for particle identification, but with high momentum.
- ECAL: the Electromagnetic CALorimeter measures the energy of electrons and photons.
- HCAL: the Hadronic CALorimeter measures the energy of hadrons.
- MUON: the last layers are the MUON walls, which are used to identify muons.

By crossing the informations provided by all the sub-detectors, we are able to understand exactly what has happened and what was produced after the collisions (see Fig. I.4).

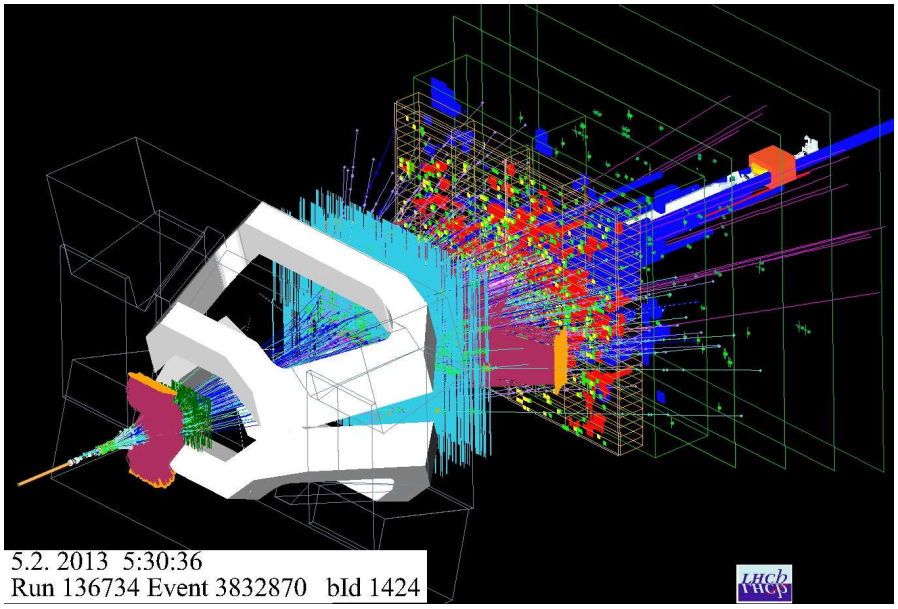


Figure I.4: Example of an event reconstructed by LHCb

Nevertheless, the frequency of collisions produced by the LHC is way too high to record all of them. It would mean to transport and store several Terabytes of data per second, which was impossible at the time of the design of the experiment, and would still be extremely challenging in the coming decade. Moreover, keeping all the data does not make any sense since we are looking for very rare events, and thus most of what we would record would be totally uninteresting and useless. The straightforward solution is filtering: we need to choose which event should be kept, and which one should be discarded. This discriminating process is called *"Triggering"*.

LHCb has two levels of triggers:

- The L0 trigger: the selection at the low level is done based on only a small fraction of the global information representing an event. That is, the decision is based on information coming from the calorimeters and the muons only. The L0 trigger is an FPGA-based electronic device, and decreases the data rate to 1 MHz, which is still too much to be stored.
- The High Level Trigger or HLT: although not using the full picture, the HLT has to partially reconstruct the event with data from all the sub-detectors in

order to perform a more detailed selection. This filtering is software based, and runs on many of the shelf servers, in the order of 1500 units. The output rate of the HLT is in the order of 5kHz.

The events are then stored on a disk based storage system. This whole chain, going from the event creation in the detector until they are stored on disk, is called *"the Data Acquisition"* (DAQ) system. With the control systems, it composes *"the Online System"*. The data sits on the disk storage for a very limited time, in the order of 30 minutes in optimal conditions, before going in the *"Offline system"*, where they are permanently stored on tape, and analyzed in depth and multiple times by physicists.

The 1500 servers used in the filtering farm represents the majority, but only a portion of the LHCb Online system. The whole detector itself is controlled with a SCADA control system relying on a computer infrastructure. And of course, all the systems listed in the section I.1 are present: storage, database, LDAP, kerberos, *etc.* In total, the LHCb online system comprises more than 2000 servers and more than 200 switches.

The LHC and its experiments are running around the clock the whole year, with a five day technical stop every six weeks and a longer Christmas break. We thus have a situation as described previously with expert on-call duties, and fewer than 12 people, including part-timers, to have all the system running through the year.

Chapter II

State of the art

Even when there is no constraint such as high availability, diagnostic and recovery assistance has always been a goal, even outside the scope of system administration — a status LED on a board or definite sound sequences coming from a device are just examples.

There exist different approaches, methods and implementations that were attempted in order to answer parts of our needs and meet the goals described in part I.3. Below is a description of those various methods, with their pros and cons, and for each of them, some implementation examples to illustrate their applications.

The simplest approach to diagnosis consists simply of constantly looking at all that could possibly go wrong in a system: this is called monitoring. There are two approaches for monitoring: active or passive. The passive monitoring means that the status of the device is not permanently updating, but rather that the device is expected to inform the monitoring system in case of status changes. The active monitoring, in turn, consists of actually fetching information from the device.

A very typical example of monitoring protocol implementing both aspects is SNMP ([Harrington 2002]). SNMP, Simple Network Management Protocol, is a standard protocol used to manage devices on IP network, by either getting information about them or setting values in their configuration. It is most commonly used to manage switches and routers. While SNMP can be used to query the information from a device, the device can also be instructed to send an *"SNMP Trap"* when something is happening, for example a temperature threshold reached. The difference between the push, the pull and interrupt principles is a very important topic in monitoring which occurs every time a monitoring task needs to be put in place.

Monitoring protocols are only one part of the needs: the infrastructure to use them properly is also needed. The way this is usually done in a professional environment is to have one server, or a set of servers, responsible for aggregating the monitoring data, collected via a dedicated network. While a very small environment could possibly rely on a few scripts for its monitoring, large structures need dedicated software. There are a great variety of such software (*e.g.* [Enterprises 2009], [Project 2013b], [team 2010], [SIA 2012]), each having pros and cons, but one of

them is really the reference open source solution: Nagios ([Enterprises 2009]).

The purpose of Nagios is to raise an alert in case of a problem, by sending an email for example. It is primarily designed for performing active checks, but also handles passive checks. It has a very modular architecture, which allows to add many functionalities that are not part of the core software, such as plotting. The configuration of the monitoring done by Nagios relies on two building blocks: hosts and services. A host normally refers to a device such as a server or a router. Services refer to all kinds of software, applications or resources like CPU or disk space, *etc.* Services are then applied to Hosts. The main status for a Host are *"Up"* and *"Down"*; for Services there are *"Ok"*, *"Warning"*, *"Critical"*, *"Unknown"*. A user defined executable for each Host and Service is then used at regular interval in order to query its status.

Software like Nagios will work out of the box for most environments, but as mentioned previously, there is a threshold after which simple setups collapse and cannot perform anymore. Then very complex distributed installation is needed, such as the one presently in place at LHCb using Icinga ([Project 2013b]), a fork of Nagios [Haen C 2011]. This allows to cope with the 40,000 Services and 2200 Hosts monitored on a 5-minute base.

However, the amount of checks executed grows up exponentially with the number of Hosts and Services, and that most of the checks executed will be useless in a sense that they will return an OK state. On the other hand, relying completely on passive monitoring is not a good option either, since it is very possible that the service or the host dies before having the time to warn anyone. So for large environment, there is no solution to reliably monitor everything without having to struggle for performances (using for example plugins like [Nierlein 2010] to distribute the work).

The other aspect which appears with classical monitoring on large scale is the massive spamming that occurs when a problem on a core system will impact many other systems, just like a power cut. This will have as a consequence to trigger a lot of alarms, making it impossible to distinguish the root cause of them.

Despite these flaws, classical monitoring solutions are still used everywhere, and are useful in many cases. However, there is clearly a need for having cleverer solutions for big environments: instead of having the status for all our sensors, it would be better to be warned only about the root problems.

The first solutions that tried to ease the work of administrators by providing them with diagnoses were based on a branch of artificial intelligence (A.I) called *"Expert Systems"* [Ginsberg 1993]. Expert Systems are said to be *"rule based"*: their diagnoses are based on sets of simple rules and conditions. One can distinguish two

main components in such an architecture:

- the knowledge base: the knowledge base contains sets of rules and conditions defined by the user. It is typically *"IF ... THEN ... ELSE"* rules.
- the inference engine: this is the part which queries the knowledge base and uses its contents in order to follow a stream of rules and finally offer a conclusion to the user.

The DENDRAL project ([Lederberg 1987]) is considered to be the first *"Expert System"* in history. This technique presents certain great advantages. The information is given to the Expert System using declarative statements, which are very easy to read, write, interpret and evaluate by a software or a human. This means first of all that the knowledge base of the software is extremely modular. Rules can be added, removed or modified at any time. The second consequence of the declarative statements is that any conclusion given as an output by the software can be explained and traced backward.This is very important if we want to understand its decision.

Unfortunately, these kinds of systems have a major flaw: in order to be useful, the knowledge base has to be constantly up to date and with exhaustive information, which is extremely demanding and difficult to achieve. Moreover, their diagnostic does not improve automatically with experience, since the knowledge base is filled in with static user declared rules.

Expert Systems have had and still have a lot of success in various domains like bio-medicine, agriculture, automobile, education ([Naser Samy 2008], [Khan 2008], [Sarma 2012], [Lehman 2006], [Khan 2011]), but their limits make them unsuitable for application with a very dynamic and wide range of knowledge, such as system administration.

In 2001, IBM introduced the concept of *"autonomic computing"* in a manifest [Horn 2001] regarding the coming problems of IT infrastructures. The manifest pointed out that the major obstacle to keep improving in the IT infrastructure is the complexity reached by those systems: it draws a parallel with the American telecom industries in the 1920's, when the lack of operators started to be critical and led to the automated switchboard. The solution IBM offers [IBM 2001] [Kephart 2003] is to give *"self-managing"* capabilities to IT systems. It relies on two fundamental concepts, which are still the basis of all autonomic systems today [Huebscher 2008]:

- The self-X: in order to be *"self-managing"*, a system must verify four properties:
  - self-configuration: auto configuration, following high level directives
  - self-optimization: system should improve its performances
  - self-healing: system detects, diagnoses, repairs

  - self-protection: against attacks, or cascade failures

- MAPE-K loop:
  - monitor: collect details from managed resource
  - analyze: data analysis and reasoning
  - plan: structure the actions needed to solve the problem
  - execute: execute the action
  - knowledge: data about the resource

This control loop is the reference model. It introduces the different working steps which lead to autonomic computing.

In principle, every element of an autonomic system (*e.g.* web services, DB cluster) should implement its own MAPE-K loop. This naturally leads to a multi-agent system: the autonomic systems are supposed to dialogue and negotiate in order to keep the whole system running.

Many implementations of this loop have been done. Some aim at being generic and are distributed as frameworks [Y. 2002], [Cheng 2006], [Kaiser 2003], [Li 2010], while other approaches are much more specific [Martin 2007],[Solomon 2010] —

- ABLE [Y. 2002]: it is a JAVA framework written by IBM which allows to develop and deploy multi-agent systems. The framework comes with libraries thanks to which one can include A.I. algorithms in the agents.
- Kinesthetics Extreme [Kaiser 2003]: it takes actions, based on sensor and XML rules. It is by definition close to the Rule-based systems.
- Rainbow [Garlan 2004]: offers a similar approach to the one of Kinesthetics Extreme.
- Logs analysis: infers interactions based on a temporal analysis of the system logs, and triggers actions based on user-defined rules.

Other approaches are much more specific —

- An autonomic behavior for a set of web services, based on a model approach [Solomon 2010].
- A adaptation of the WSDM specification to make it match the self-managing principles [Martin 2007].

In any case, there are two situations: either the source code of the software is available, in which case it is possible and better to adapt it to give it autonomic capabilities; or the source code is completely hermetic.

Most of the papers concern the first situation, which covers various cases:

- Writing new software following some principles [Karsai 2001], [Poirot 2008]
- Adapting the code by following principles of aspect oriented programming [Chan 2003], [Sadjadi 2005]

Some papers offer a method that they call *"non intrusive"*, which only works for Java programs, since their solution consists of injecting code at runtime via the Java Virtual Machine [Chan 2003]. This technique relies on aspect oriented programming (AOP) [Kiczales 1997]. Roughly speaking, AOP consist of splitting the program's code and logic according to so-called *"concerns"*. By using dedicated Java frameworks like AspectJ [PARC 2001], it is then possible to develop the monitoring of an existing application as a concept, without modifying the original code, and merge both at the bytecode level.

Papers about the second case are very rare because the only grip we can have on software are those offered either on purpose via API, or via security issues. That is, the only interaction points with a software are the interfaces (either command line or graphical), the logs output (more or less verbose, and not always relevant) and less frequently an API which gives you an interface easily usable in your own code. It is thus very hard to have a generic approach.

It is worth to mention one other approach, which consists of giving the users a new language in order to describe their system [Cheng 2006],[Trencansky 2006].

Other papers offer a more *"systematic"* approach of the problem. It is the case of [Dolev 2008], which suggests to simply periodically reinstall the whole system, or alternatively, to develop an operating from scratch system whose primary constraint is the stability of the software.

Another common point of the different approaches is that they are *"model based"* or *"architecture based"* (see [Brambilla 2012] or [Lano 2009] for an illustration of those concepts in software development). This means that the autonomic mechanism has a simplified representation of the system, which allows to see whether the system behaves as expected (a monitoring system) and eventually to test changes on the model before applying them. This methodology is also used in systems based on the control theory and attempts have been made to mix control theory and autonomic computing [Karsai 2001],[Diao 2005],[Stoilov 2010].

Eser Kandogan [Barrett 2004] has analyzed the characteristics an autonomic system should have, from the point of view of system administrators. The main points are:

- to be able to know what the system is doing, what it has done and what it wants to do

- a command line interface should be available for recurring tasks
- a graphical user interface is more suitable for exceptional cases

In conclusion, one can find several common points:

- In one way or another, there is a rule based system which makes the link between measurements and actions
- Need to describe what to monitor
- Need to describe how to monitor
- Need to specify the rules to trigger an action
- Need to specify what to do in case of problems

The aim of this work is to improve the last four points quoted above.

Chapter III

LISA: LearnIng approach for System Administration

Contents

As seen in the state of the art, most of the solutions offer an approach which is fully dedicated to a given problem. One will always have much better results if the autonomic functionalities are taken as a requirement when designing the solution, rather than adding them afterwards. However, LHCb is running a great variety of software:

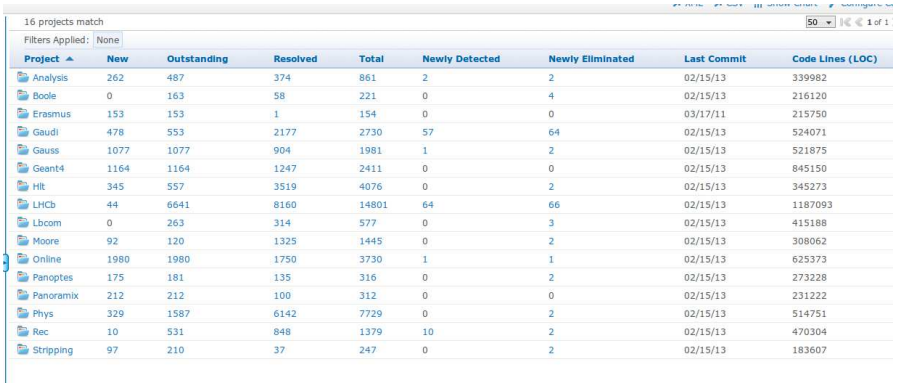
- many are in-house software, counting more than a million lines of codes (Fig. III.1).
- others are open-source or proprietary software, like for the databases, websites, and most of the infrastructure software stack

While it is clearly impossible to rewrite or modify all those software, the aim is to have a tool as generic as possible.

III.1 Restrictions

Having a system which answers all our needs in all situations is out of reach, and this is why we need to impose some restrictions.

The first of them concerns the operating system. While the LHCb Online system is running several versions of Windows and Linux, the Windows variety



16 projects match

Filters Applied: None

Project	New	Outstanding	Resolved	Total	Newly Detected	Newly Eliminated	Last Commit	Code Lines (LOC)
Analysis	262	487	374	861	2	2	02/15/13	339982
Boole	0	163	58	221	0	4	02/15/13	216120
Erasmus	153	153	1	154	0	0	03/17/11	215750
Gaudi	478	553	2177	2730	57	64	02/15/13	524071
Gauss	1077	1077	904	1981	1	2	02/15/13	521875
Geant4	1164	1164	1247	2411	0	0	02/15/13	845150
Hlt	345	557	3519	4076	0	2	02/15/13	345273
LHCb	44	6641	8160	14801	64	66	02/15/13	1187093
Lbcom	0	263	314	577	0	3	02/15/13	415188
Moore	92	120	1325	1445	0	2	02/15/13	308062
Online	1980	1980	1750	3730	1	1	02/15/13	625373
Panoptes	175	181	135	316	0	2	02/15/13	273228
Panoramix	212	212	100	312	0	0	02/15/13	231222
Phys	329	1587	6142	7729	0	2	02/15/13	514751
Rec	10	531	848	1379	10	2	02/15/13	470304
Stripping	97	210	37	247	0	2	02/15/13	183607

Figure III.1: Screenshot of the LHCb software stack

represents less than 5% of the total. The efforts will thus be focused on the great majority, that means on Linux. For the time being, Windows machines are completely ignored. The reason for it is that the philosophy of both operating systems is completely different, and so is the diagnosis of a problem. However, the software design and our algorithms contains, in principle, enough abstraction layers to consider adding Windows diagnoses in the future. This aspect will not be addressed at all in this work.

Another aspect of diagnoses and recoveries has been considered and discussed for a while, before discarding it: network diagnoses. Network diagnoses and recoveries are very special because they require that you look at the problem from different perspectives and different points of view in order to understand it. This would have required a highly distributed software, which brings many other complications, not only in the development techniques but also in the algorithms, not mentioning considerations like the split brain situation. The split brain situation occurs when two or more parts of a distributed system lose communication with the others, and that each of them considers that it is the only part running: this leads to inconsistent situations, which are very often unrecoverable and fatal.

To summarize, our work will focus on the diagnoses and the recoveries of systems running on the Linux operating system, and leaving the network problems aside.

III.2 Diagnosis and recovery of a problem

Before being able to design a software able to diagnose and recover problem on a system, it is important to understand how a human expert would do it: after all, the software tries to mimic his behavior. Let's split an ideal diagnosis and recovery process step by step:

- Alerting the expert: the very first step is to warn the expert about ongoing problems. Most of the time, this will be done either by a classical monitoring system as described previously, or, more likely, by an unhappy user.

- Filtering the information: whether it comes from the monitoring system or from the user, the *"problems report"* done previously contains noise, that is, useless information. The useless information is usually about unavailable services reported as problems, while they are just consequences of an underlying problem. From the monitoring, the noise will show up as additional alerts and emails; from the user, the noise is usually a result of his poor knowledge of the system and his emotions of not being able to use his favorite software at this very moment. So the expert has to filter the information, and recall how the investigated services work; that is, the different components, their behavior, their dependencies and interactions.

- Making hypothesis: thanks to his knowledge about the system infrastructure, the expert can establish a list of possible culprits to explain the ongoing problems. This list will be sorted according to the dependencies between the components. There is yet more data that an experienced expert can use to sharpen his hypothesis: the likelihood of a given component failing. This is a very important piece of information. It allows to accelerate the diagnosis by investigating the most likely cause of problems first. Once the list of hypothesis having been established and sorted, the expert will test them one by one, until he finds the correct one.

- Fixing the problem: once the problem has been tracked down to its root, fixing it normally requires a simple, specific step. However, this specific step hardly comes alone: it will indeed fix the root problem, but extra actions might be required in order to take this fix into account. As an example, think of a configuration file which is read at the start of a program. If an odd behavior of a software is noticed, and tracked down to a mistake in the configuration file, fixing the file will not be enough — a restart of the program will also be needed. Similarly, just before taking an atomic recovery action, the expert has to think of all the other actions that it implies.

- Updating the problem's status: once the expert believes he has diagnosed and fixed the problems, it is in good taste to cross check that the intervention was indeed successful. This is done by having a look at the monitoring status, or enquiring about the previously unhappy user. In any case, at the end of that step, one should have an up-to-date list of problems. Ideally, it will be empty, otherwise one goes back to the first step.

Of course, this description only works for ideal cases, and one has to take into account all the complications that can occur, such as concurrent problems, wrong diagnostics, inefficient recovery action, *etc.*

The software that we want to design aims at following similar steps.

III.3 Software running on Linux

This part is not intended to be a detailed description of the Linux operating system, but rather to stress out some of its characteristics that will be relevant in our context. Also, since our software is only meant for Linux, one can fully exploit its specificities without thinking too much of the compatibility with other operating systems.

Linux [Proffitt 2009] is an open source operating system compliant with the POSIX norm. There are many Linux distributions, but the one used at LHCb and at CERN is Scientific Linux Cern (SLC), which is based on RedHat Enterprise Linux (RHEL). LHCb is currently running with two versions of this OS: SLC5 and SLC6.

One of the defining features of Linux states that *"Everything is a file"*. This means that any resources can be accessed through the filesystem with a unique interface: documents, hard-drives, modems, keyboards, printers, network socket or even memory.

A very special folder in the Linux OS is the *"/proc"* folder. This folder, although looking like any other folder, is actually not stored on the disk, but is in memory, and contains information about the Linux kernel and the running processes. Processes are programs that are being executed. The kernel, illustrated in Fig. III.2, is the core part of an operating system which manages the resources, provides an abstraction layer between the software and the hardware, and offers facilities such as inter-process communication.

What this means is that the user is offered to access all the information regarding the hardware, the system and the processes, via a unique interface: a file. This is a very powerful tool, because the data possible to retrieve from these files are precisely defined and formatted by standards.

Any service can be seen as one or several applications working together. An application itself is nothing but one or several processes using files (as input or output).

In consequence, regardless of the way the software is implemented, or whether it is an in-house application or a proprietary solution, it can always be described as a set of elementary entities with well defined attributes, that is processes and files. And this is exactly what we are looking for in our software.

There is of course a flip side to this approach, which is that one has to work at

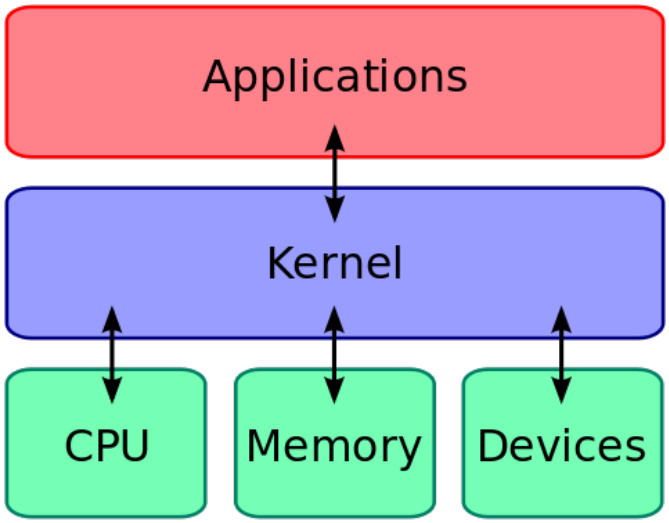


Figure III.2: Illustration of a kernel layout

a low level, while using high level directives is much more natural and easier. Also, one will remain blind regarding the implementation of the software, and about the process information stored in memory only. Describing applications as a set of files and processes can be quite complicated and cumbersome, but it is the most generic and least invasive approach that is available under Linux.

III.4 Adaptation of the MAPE-K loop

The good results obtained by the various implementations of the MAPE-K loop suggest to follow this principle. [Roblee 2005], [W. Strickland 2005], [Xu 2005], [Schmerl 2002], [Sterritt 2005], [Y. 2002], [Kon 1999], [Hofmeister 1992], [Magee 1989], [Kaiser 2003], [Parekh 2006], [Y. 2002] or [Cobleigh 2002] are some examples of successful implementations. Also, it is quite close to the *"diagnostic and recovery procedure"* previously described .

The existing solutions based on the MAPE-K loop use one loop per system or service, and then rely on multi-agents theory to make them communicate. The communication between the different loops is very important and cannot be avoided, in particular at the *"Plan"* stage. Synchronization is a must to avoid taking contradictory actions. The advantage of this method is to split the complexity of the knowledge between various systems. Nevertheless, the complexity induced by the communication between the systems grows exponentially, and quickly becomes unmanageable.

The innovative approach that we offer here is to have a unique MAPE-K loop, shared by all the systems.

First of all, this allows to get rid of the complexity of the multi-agent communication, which is non-negligible in a big environment such as LHCb. Multi-agents involve conflict resolutions and cooperation problems that are still fields of intense research ([Lupu 1999], [Kamoda 2005], [Ananthanarayanan 2005], [Jennings 2000] for instance). The cost of it is that all the knowledge is centralized, which is not necessarily a bad thing per se.

The second and most powerful advantage of this approach is that it allows to discover dependencies between the various systems. Because all the systems are managed by a single loop, one can observe the influence an action on one service has on the others. So while the user is able to inform the software of rules such as *"Service A needs Service B"*, the software would be able to discover itself such dependences, either forgotten by the user or simply not known to him. This is a very useful feature because this means that the software is able to infer information about the infrastructure, just as an expert would do.

The second modification that we propose concerns the Monitoring part of the loop. The traditional approaches suggest having the monitoring as a part of the loop and ideally as a part of the system itself. This means to have some kind of sensors implemented through the application. The advantage of this is the precision of the monitoring: the decisions that are taken by the MAPE-K loop, specific to that system, are based on many inputs. The drawback is that we do not always have the possibility to implement such sensors, and that it would anyway not be efficient when working with a single MAPE-K loop for all the systems: this brings us back to the classical monitoring solutions and their scalability issues. Finally, this method ties monitoring and internal working together.

In the description of a standard *"diagnosis and recovery procedure"* described above, the alerting is done by an external source: a software or a user, but not the system itself. There is another major difference with the previous approach: the monitoring implemented directly into the software will give low level information, whereas the user reports high level information, that is whether a service works or not. This has two direct consequences: the administrator has to monitor and check much fewer sensors, and the monitoring is completely independent of the internal working of the software. For that reason, the second modification we want to apply to the MAPE-K loop is to outsource the monitoring and have high level information. In the same way that the role of the expert is to fix the problems reported by the user, we want to have our software to be notified of problems.
Another reason for not implementing monitoring feature is because there is already a great variety of software designed for that purpose, and doing it very well. Finally,

having such a high level and independent monitoring allows to have several sources of information: the complaints may come from users' direct inputs, from ticketing systems, from classical monitoring solutions, *etc.*

III.5 Reinforcement learning

The first major flaw of classical monitoring and expert system techniques is that they are very static: they are only able to do what the user configured them for. Namely, a classical monitoring system will be only testing the probes users have specifically mentioned, and an expert system will only follow the rules that users have given it. As emphasized in section II, this requires to keep their configuration up to date in order to have a useful system, which can be extremely cumbersome. We have also seen that large configurations cause problems in terms of performance.

The solution we are looking for should overcome those limitations by being more dynamic and perform in a large environment. The performance problem can be addressed by having a methodology which is cleverer than exhaustive checking. The dynamic aspect requires that the information at the disposal of the software is modular, and that the software should be able to modify it, or even better, to infer new information.

These expectations lead us to consider reinforcement learning techniques. Reinforcement learning is an area of machine learning, in which the goal is to learn from experience what to do in various situations, in order to optimize a numeric reward (see [Sutton 1998] for an introduction). The range of application of such algorithms is extremely wide: from network routing to cell phone channels allocation, to games intelligence, to combinatorial searches, robotics or even dialog strategies ([Vidal 2003], [Buşoniu 2006], [P.R.J. Tillotsona 2004], [Riedmiller 2009]).

A faster diagnostic can be obtained by reducing the amount of components that are checked, and by testing them in an appropriate order. The reinforcement learning can help by optimizing the component sorting, based on previously encountered situations. This exactly mimics the approach a human expert would have. Note that this also solves the performance problem, since much fewer components are checked, and there are good chances to find the faulty one among the firsts tested.

The single MAPE-K loop principle described previously, together with reinforcement learning techniques, would allow to discover new dependencies between systems, i.e., new rules, and thus answer the dynamism requirement.

III.6 Shared Experience

While answering our needs in terms of functionality, the reinforcement learning techniques have a weakness in the application phase — in order to be efficient, they require a lot of training. This is obviously a problem in an environment like LHCb, in which voluntarily creating problems is out of the question and which is too big and heterogeneous to be efficiently simulated.

The answer we have to this problem is the *"Shared Experience"*. The idea is to share the experience gained by the various similar systems as much as possible, under real conditions, thus minimizing the learning phase. If we were to draw a parallel with a human expert, one could consider the following case: if an administrator is able to maintain, diagnose and recover problems on a given web server, there are very good chances that he is able to extrapolate his knowledge on another web server. Our aim is to integrate this principle in our solution, and thus gain a lot in efficiency.

The principle of grouping similar systems has another advantage for the system administrators, which is that it dramatically reduces the description and configuration workload compared to traditional systems. Indeed, for large environments, the configuration required by classical monitoring system can be extremely long. As an example, the Icinga configuration to monitor the LHCb Online environment comprises about 34,000 lines. By extending the sharing logic up to the configuration, one can reach a point where a system would be described once, and all similar systems would refer to that primary one. This would save a lot of time of the system administrators.

III.7 Convention over Configuration

Now we need to remember that we want to lighten the work of the system administrators, and not give them extra workload by forcing them to write huge amount of configuration lines. An approach that normally feels comfortable for the users is called *"Convention over Configuration"* (CoC) as described in [Miller 2010].

The principle of CoC consists of assuming a default choice in all cases except those specified by the users. This decreases the configuration workload of the users a lot because they only need to configure what is not standard. This technique is actually mostly used by frameworks to ease the work of the developers. Famous examples are Java Spring [Rod Johnson 2009], Ruby on Rails [Team 2013] or PHP Symfony [Labs 2013] .

Recall from III.3 that the basic blocks of the software will be files and processes, and that there are standards to define their interfaces or the interactions one can

have with them. We want to exploit this fact in our software to enforce the CoC approach.

Note that this method is absent-mindedly used by human experts. For example, if an expert tracks down the problem to a Linux service not running, that he does not know, but that he knows should be running, then he will assume — correctly in most of the cases — that it can be started like any standard Linux service.

The CoC principle is used through all the software, and is the main guarantee that the first goal listed in I.3 is not forgotten, that is to reduce the work of the system administrators.

III.8 Diagnosis & Recovery

Providing the expert with the faulty component is already a great help, but we target even higher: we want to provide him with a recovery solution. In order to achieve this goal, two sources of information will be used:

- Default actions: once again, using standard entities (Files and Processes) and CoC gives the opportunity to provide the user with a recovery solution that should work in most of the cases. As an example, if a file does not belong to the right owner, there is only one way to change this, and this is the solution we will be proposing; or if a service is not running, there is only one standard way to start it.
- Custom actions: in some cases, the standard procedure does not work. For this reason, when a problem occurs on a given entity, the user will be prompted with the actions that have been taken on it in the past, with comments from the expert that took the action.

In section III.2, the reasoning of a human expert to diagnose and solve a problem was described. When the root problem is found, correcting the mistake is usually a simple action, but it might require a full chain of events in order to take this modification into account. This step should also be part of our solution.

Proposing to the user the action to fix a file or a process is not enough, one needs to provide him with a full procedure that will bring the whole system in the expected state. Unfortunately the software is not able to guess the procedure. It is easy to understand why with an example: a web server such as Apache will read its configuration files only when it is started, but it will read the files it serves every time there is a request for it; so a modification in the configuration file will require a restart of the Apache server, while a modification in the pages it serves does not. And there is no way for a software aimed at being generic to be able to

distinguish whether a modification on a file requires the restart of the service or not. However, what can be done is to build the full recovery procedure from several atomic pieces of information given by the user. An atomic information will have the form *"If <action X> is taken on <component A>, then execute <action Y> on <component B> <before | after>"*. By putting together all these statements, the software is able to give all the steps required. Notice that executing an action on an entity might require to take other actions after it (like restarting the process after changing the file), or before (like stopping a process locking a file before modifying this file). While this seems to be just a detail, it greatly improves the complexity of the algorithm that compute the full procedure as we shall see later.

III.9 Summary

To summarize, our software will have the following characteristics and be using the following approaches:

- Focus only on Linux system, and not attempt to deal with network problems.
- Be totally non intrusive and as generic as possible by being based on files and processes as basic blocks.
- Perform no monitoring, but rather wait to be informed of problems by external sources.
- Adapt the MAPE-K loop, and use a single loop instance to look at all the problems at once, and not treat them separately. This will allow the software to spot the dependencies between the various systems.
- User Reinforcement learning algorithms to improve the diagnostic speed and help in scalability by reducing the amount of components that are checked before finding the faulty one.
- Reduce both the learning phase of the learning algorithms and the description workload of the users with the Shared Experience principle.
- Using Convention over Configuration contributes in reducing the configuration work of the software.
- The software will offer default recovery solution with the full procedure for it to be efficient, as well as information regarding the previous situation encountered on the same problematic entity.

This methodological framework should answer all the needs we have listed in the section I.3.

The principle of expecting information coming from a third party source suggests active learning, which is a specific branch of machine learning. The idea of

active learning is that the so-called *learner* will learn faster if he can query an *oracle* who answers his questions. The role of the algorithms is to select unlabeled, that is unclassified, instances to be labeled by the oracle. There is a great interest in this when one can very easily have instances, but labeling is an expensive operation, like in speech recognition ([Zhu 2005] mentions a factor 400 between the times needed to get instances and label them), information extraction ([Craven 2008] gives many examples) or classification and filtering. Research focuses on how to improve the choice of the instances to be labeled by the oracle: [Tong 2001], [Tong 2002], [D. A. Cohn 1996], [Monteleoni 2006] or [Olsson 2008] are just a few examples of the many approaches.
The methods we decided to use can be put in parallel to the active learning:

- Regarding the monitoring, all the information is expected to be given by the user, which means that we do not select the information we request. This is not in concordance with active learning.

- However, the potential failures to probe are chosen among all the known failures, and the verification done on the remote servers is the oracle queried.

- The user can be seen as an oracle who is asked to confirm the dependency rules.

The *"active feature acquisition"* approach is a special case of active learning. The assumption behind this method is that it is possible to request more information about an instance at a certain cost. The goal in active feature acquisition is to choose which information is the most relevant to request. [Zheng 2002], [P. Melville 2004], [M. Saar-Tsechansky 2009], [Greiner 2002], [Charles X. Ling 2004], [X. Chai 2004], [Sheng 2006] and [Ji 2007] all offer various methods to choose which information to request.
As will be described in this document, the software actually queries the remote servers for information, and not directly for a label. Also, based on the returned information, the diagnosis will take different directions. The cost to bear is the time it takes to retrieve the information. Thus, one could draw a parallel between the queries to the remote servers and the *"active feature acquisition"* approach.

To conclude with active learning, one could see similarities between our approach and the *"active class selection"* ([Lomasky 2007]). While Active learning usually assumes that instances are free and labels expensive, *"active class selection"* assumes the opposite. Querying the remote servers or waiting for monitoring updates takes much more time than comparing the returned values and so giving a label, so there are similarities from that perspective.

One could also note that our approach matches the context of the on-line learning (see [Cesa-Bianchi 1997], [Littlestone 1994], [Vovk 1990] and [Sleator 1985] for details on the on-line learning). On-line algorithms consist of the following steps:

1. The learner receives an instance to label
2. The learner makes a prediction on the label
3. The learner gets a feedback which is the real label

The goal of those algorithms is to improve the labeling thanks to the feedback (see [Kanal 2000] for the classical Perceptron example). The diagnosis algorithm that we developed, detailed later in this document, has common points with the on-line learning:

1. The software is notified about a problem
2. The software makes a prediction on what the root cause is
3. The software gets a feedback by querying the remote servers to confirm its hypothesis

The next section focuses on the actual algorithms and implementation aspects.

Chapter IV

Phronesis

Contents

We named our software *"Phronesis"*. Phronēsis or φρόνησις is the Greek word for wisdom or intelligence. We will first describe the software from a global point of view, explaining the different components, the packaging and the database schema. In the second part we will have a more in depth look at all the tools required to use the software. The last part will then focus on the Core part, in which all the algorithms are implemented.

IV.1 General considerations

Before looking at the actual algorithms and implementation details, it is important to understand the global picture of the application.

IV.1.1 Basic components

In this part, we will describe the various components with which the software deals. This means the internal representation of the various information the user can give.

The very basic entities we are working with are Files and Processes.

A file in Linux has the following properties, defined by the POSIX standard [IEEE 2013]:

- Filename: a file is identified by a name which is its full path to access it.
- Owner: the username to whom the file belongs.
- Group: the name of the group of users with which the file is associated.
- Permissions: there are three types of permissions which are *"Read"*, *"Write"* and *"Execute"*. For each file one defines the permissions for the owner, the group and the others — so three sets of permissions. The way they are represented is with three octal numbers, one for each set of permissions. The octal number represent groups of three bits in the permission field: Read's bit has the value 4 in octal, Write's bit has 2 and Execute's bit has 1. For example a file with the permissions *"754"* means that the owner can Read, Write and Execute, the group can Read and Execute, and all others can only Read.
- Attributes: in addition to permissions, files have attributes which set options on the filesystem. Each attribute is represented by a bit. Typical attributes are for example *"immutable"*, *"append only"*, *"undeletable"*.

A Linux process has the following properties:

- Process name: it is the full path to the executable and all the arguments given.
- PID: the PID is a unique number used by the system to identify the process.
- User: the user id (UID) to whom the process belongs. A process inherits the permissions of the user. If a user has the permission to read a certain file, all the processes owned by that user will have the permission to read that file. Most of the time the process belongs to the user who started it.
- Limits: the kernel will enforce certain limitations and quotas on every process. For example, there may be a maximum number of files opened by the process or a maximum of memory consumed by it. Note that the limit can be unlimited.
- Parent: a process has a parent process, which is the process that started it. Absolutely all the processes have a parent, except the process with a PID of 1, which is the root process.
- Environment: every process has a so-called *"environment"*, which is a list of key-value pairs colloquially referred to as environment variables. Those variables are inherited from the parent process, given to the children process, but can be modified by the process itself. A typical environment variable example is $HOME, which contains the path to the home folder of the process user.

In order to work with Files and Processes, a class called *"Agent"* was developed for each of them. Those classes are used to represent the properties listed above, as well as other useful information for diagnoses and recovery. The Agent representing

Files is called *"FileAgent"*, the one representing Processes is called *"ProcessAgent"*. Each of them is responsible for checking that the actual properties of the associated file or process correspond to what the user expects.

In addition to the file properties, the FileAgent contains a few more attributes:

- MD5 checksum: while not a property of a file per se, a file has a unique md5 checksum. Thanks to this property one can make sure that a file is exactly as the user expects it to be.
- Content: as for the md5 checksum, the content of a file is not a real property. A user might want a certain content to be present in the file, without giving attention to the rest of the content. This attribute actually contains user defined regular expressions that the file content should match.

The ProcessAgent also has extra attributes:

- max CPU: it may happen that a process — for one reason or another — starts consuming computing power beyond what is reasonable, preventing the server to do anything else. Sometimes this might be an expected behavior, but in order to take into account cases in which a process should not be so compute power intensive the user has the possibility to define a maximum threshold after which the process is considered to have a problem.
- max memory: similar explanation as the max CPU, but for the memory consumption.
- Command: it is frequent that the process name, which contains the executable and all the arguments, is not the command the user uses to start the process. Often, there is a wrapper that the user executes and which abstract the complexity of starting many processes with many arguments. This Command attribute contains the information of how the process was actually started.
- Service: a service in Linux is a process or a set of processes that are running in the background to provide certain functionalities. For example a database or an ssh server. At the low level Services are standardized wrapper scripts described above with a defined uniform manner to start, stop or restart the processes. The service attribute of the ProcessAgent contains the name of the service used to manipulate the process.
- Multiplicity: it is not rare to have several instances of the same process. For example, a web server will start as many processes as the amount of clients it has to serve. Usually there is an option for maximum number of processes, that the user can set. In case there is not, too many processes can kill the machine. The multiplicity is the maximum number of simultaneous identical processes that should be running on the machine.

Another Agent was defined to represent the status of the machine and its environment as a whole: the *"EnvironmentAgent"*. It contains information about global values, such as the free disk space or the free memory, which are system wide and impact all the services running on that server.

The properties of the EnvironmentAgent are the following:

- Maximum Load: the load is a measure of how much the system is used. In simple cases, a small load value means that the server is mostly idle, while a high value means that the server is being intensively used. A too high value can cause all the applications to react very slowly, and sometimes become unusable. This property allows the user to set a threshold after which the machine is considered to be overloaded.
- Maximum memory: a system whose memory is full will have the same impact on the applications as an overload one. Sometimes worse. This property sets a threshold in the memory usage.
- Maximum swap: in case the machine runs out of memory, it can move rarely used data to disk until it is needed again. This operation is called swapping. The user can set a maximum swap space occupancy threshold.
- Fstab: the fstab is a special file which defines what filesystem, local or remote, the system should mount. While it would have been possible to let the user configure a FileAgent to check the content and properties of the fstab, it makes more sense to abstract it from the user and just let him define what filesystem he expects to be mounted and with what options. Also, the fstab should always belong to a special owner and group (called root) and have the permissions set to 644. Note that abstracting all this from the user corresponds to the CoC principle.
- Disk space: threshold on the maximum disk usuage
- Inode usage: in a filesystem, every file is identified by an inode with a unique id. When a file is created, the filesystem assigns an inode to it. If the filesystem runs out of inodes, then no files can be created anymore.

To follow the CoC principle, all these thresholds have default values.
Because of unusual behaviors some file systems may have, the user also has the possibility to specify a list of file system mountpoints to ignore.

The last agent that was defined is called *"FolderAgent"*. It represents a folder, that is a container for files and sub-folders. At the lowest level a folder is just a special file, whose data is the list of files it contains. The FileAgent can be used to check the basic properties of a folder, like its owner or the permissions. The reason for having a FolderAgent is to emphasize the *"container role"*: all the contents of the folder must match defined properties. This gives some flexibility in case

files are often created or removed from a folder, but must always satisfy certain conditions. For example: all the files in this folder must be owned by the user 'root'.

The properties of the FolderAgent are the same as those for the FileAgent except the md5 checksum, which does not make any sense here.
In case the user does not want to check all the contents of a folder, there is a possibility to filter the content names according to regular expressions.

All of these are Agents for describing the basic blocks that are files and processes, as well as describing folders and environments. Now one needs to describe an application or a service as an aggregation of all those Agents. This is the role of the so-called *"Coordinator"*. Agents and Coordinators are both of the abstract type *"MetaAgent"*, as seen in Fig. IV.1.

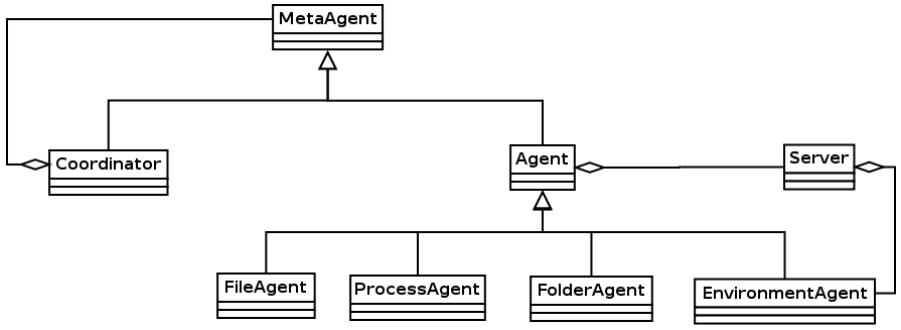


Figure IV.1: UML class diagram of the MetaAgent class

A Coordinator contains a list of MetaAgent children that belong together logically. For example, one could put together a process and its configuration file. The Coordinator is responsible for checking the children in the appropriate order. This order comes from the reinforcement learning algorithms that we will describe later.

Of course, processes and files run on a given server, which the user needs to define. A very simple class called *"Server"* contains a logical name as well as its network address. A Server has always an EnvironmentAgent associated with it. In fact, the user never has to define an EnvironmentAgent. The user defines a server with certain properties and limits and the EnvironmentAgent is defined automatically behind the scenes. This approach is much more natural.

The philosophy of Phronesis is to consider the services as central, not the servers. It is the service provided by a Coordinator which matters, not the server on which it runs. The user normally defines all the MetaAgents he needs, and states that it runs on a given Server. There is however an exception to this. Sometimes

it is necessary to consider that some Coordinators are tightly coupled to a server. This happens when the coordinator does not represent a *"service"* (like a database, a web service, *etc*) but more a configuration of the Server (such as the LDAP client configuration for example). These kinds of Coordinators are called *"Attached Coordinators"*, and are treated like the EnvironmentAgents. I.e. a representation of the Server configuration.

The Server and the MetaAgent classes allow to define all services, but not their interactions and dependencies. For this, another type of information called *"Rules"* is necessary. There are two types of Rules:

- *"Dependency Rules"*: a dependency rule simply states that a service/Coordinator A needs a service/Coordinator B in order to be functional. There is no more technical information, like on which component it depends. This information is enough in order to diagnose problems in the appropriate order. If A and B are down, starting with A is pointless since it relies on B. Those rules can be user defined, but also discovered by the software.

- *"Recovery Rules"*: the recovery rules, also called *"Trigger"*, are the atomic recovery information mentioned in section III.8: *"If <action X> is taken on <component A>, then execute <action Y> on <component B> <before | after>"*. These rules are user defined only. The actions are detailed in section IV.3.1.ii.

The user now has the possibility to describe the components of its environment, the dependencies between the different services, as well as the atomic steps involved when performing recoveries. Note that the user does not need to specify how to get the information or how to treat it. Everything is hard coded in the software, since the software relies on standard manipulations.

Finally, it might happen that part of the system is not in its normal state, but that it is an expected behavior, for example during maintenance periods. To cover such cases, a *"veto"* functionality was added. A veto is just a name associated to a boolean. A Coordinator can have several associated Vetoes. If one of these Vetoes is true, the diagnosis stops and the Coordinator is ignored. Note that the value of the Veto is never measured, but always given by the user.

We just described the approach we have in configuring the software. The actual configuration tool is described in section IV.2.1. We will now take a look at the different packages of the tool we developed.

IV.1.2 The packaging

The software has been divided into several packages, each having a dedicated purpose. As we will see, they sometimes need to interact with each other. We will

now describe the partitioning of the software. Each package will have a dedicated section further.

- Database : all the information used by the software is stored in a database, whose schema will be detailed in section IV.1.3.
- *"Configuration"* package: The first tool the user will have to use. It consists of a compiler which reads and interprets the configuration files given by the user, and writes the data into the database. The advantage of having a package dedicated to this task is that changes in the syntax described in section IV.2.1.ii impact this package only.
- *"Core"* : the second package is the *"Core"* part of the software, in which all the diagnosis and recovery algorithms are implemented. It represents the central part of our solution. This package uses the information stored in the database, expects inputs regarding the ongoing problems and queries the *"Remote Agent"* Package.
- *"Remote Agent"* Package: it has to be deployed on all the servers that the user wants to analyze. Its role is to answer the queries of the Core, which typically consist of information about a file, a process, a folder or the environment. It can be considered as a simple toolbox and a change in the algorithms used in the Core should not have any impact on this package.
- API: it exposes Core functionalities to the outside, so that interfaces (command line tools or GUI) can be built on top of it. The great advantage of this approach is that if there are technical changes in the Core, the API internals can be changed and always offer the same interface to the tools built on top of it.

IV.1.3 The database

The database is a central part in our system. The software will store in it:

- The configuration given by the user
- The results of the reinforcement learning process
- The history of previous diagnoses and recoveries

It will be filled by the Configuration package for what concerns the configuration given by the user, and by the Core package for what concerns the history and the algorithms' results.

The full schema and its description are shown in appendix A.

It is important to note that while all the tables are linked together using foreign keys, all the history tables are decoupled from those keys. The tables refer to the MetaAgent instances by their respective agent names and use these strings as part of their index. The reason for this is that one may want to keep the history of a particular service even after the configuration was removed. There is a pitfall which is that if a new service has the same name as a previous service that was already deleted, the history of the original service will be associated with the new one.

We shall now see more in depth the different packages and tools used to make the whole software work.

IV.2 Tools

IV.2.1 Configuring the program

As already mentioned, the user needs to provide our software with a minimum number of parameters. Section IV.1.1 explained the concepts given to the user to express this information. We will now look at the details of the configuration by first explaining how to apply the Shared Experience principle at the configuration level. We will then describe the grammar we have defined and finally give technical details about the compiler we implemented.

IV.2.1.i Adapting the object model

We have seen in section III.6 that sharing experience provides advantages for the reinforcement learning algorithms, as well as for the user. The Shared Experience principle consists in sharing experience between similar services. In order to do so, the services need to be classified, i.e. we need to state that *"a given service is of a certain type"*.

It turns out that the Object Model has very similar concept with inheritance, and this is exactly what we want. In the configuration, the user can define a so-called *"Abstract Service"*, and then define real services that inherit from the Abstract one. All the services so defined will share the properties defined in the Abstract Service. All the improvements brought by the reinforcement learning algorithms on one service will benefit all others.

Let's take again the website example, that we describe with an UML diagram in Fig IV.2. A very general website can be defined as an aggregation of:

- A httpd process to serve the requests
- Configuration files to set the properties of the httpd process
- Data files to be served

Using the terms seen in IV.1.1, we have a Coordinator Website containing:

- A ProcessAgent httpd
- A Coordinator WebData
- A Coordinator WebConf

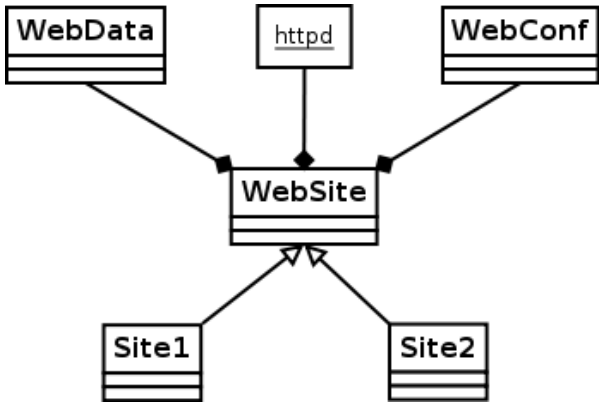


Figure IV.2: Websites described using UML diagram

Note that if all the configuration files are in the same folder and share the same properties, a FolderAgent could have been used instead of the Coordinator. The same for the data files. However the Coordinator is much more general, and this is what is wanted for an Abstract Service definition.

While the WebData and WebConf Coordinators are empty, some properties of the ProcessAgent can already be set because they should be the same for all websites.

Finally, one can define a Coordinator Site1 and state that *"Site1 inherits from WebSite"*. All that needs to be done is to detail the configuration and data files and associate it with the proper Coordinator. There is no need to mention anything about the httpd process, since it is already known by WebSite.

Note that contrary to some implementations of the object paradigm, such as the C++ one, multiple inheritance is not supported. For instance, it would be possible in C++ to define an abstract class Pet and an abstract class Mammal, and then define a class Cat which would inherit from Pet and Mammal. This is not supported.

Multiple layers of inheritance are supported. For example, it is possible to define a class Cat that inherits from class Pet that inherits from class Animal.

IV.2.1.ii Grammar

We will now describe the grammar used by the user for the configuration.

The definition of any entity starts by using the keyword describing the type of the entity (like Server, FileAgent, *etc*) followed by its name. The name of the entity is what is used through all the configuration to refer to it. The definition of its properties is contained between the two curly braces and each property is set using the convention: *" <propertyName> → <propertyValue>;"*. While some properties are mandatory (signaled with the letter *"M"* when listing the properties of an entity type), some are optional, in which case they are either ignored, or set to a default value. Except for a very few exceptions, an attribute appears only once per definition.

The simplest of all is to define a Server. This is done the following way:

```
# A comment
Server <serverName> {
    address -> <serverAddress>;
}
```

The following attributes can be defined for a Server:

- address (M): A DNS name or an IP address. It defines the network address of the server.
- maxLoad: A floating point number. It defines the maximum load tolerated on the host before considering it responsible for a problem. 0 corresponds to a completely idle host, and everything above 1 means that some queuing in the execution of the process is accepted. The number is normalized with the number of cores in the machine. The default value is 2.
- maxMemory: A percentage. It defines the percentage of memory allowed to be filled in. The default value is 100.
- maxSwap: A percentage. Similar to maxMemory. The default value is 99.
- mountpoint: A line that is supposed to be in the fstab. The mountpoint attribute can be used several times in a Server definition.

Because constraints on the Server, such as the maxLoad or the maxMemory, can be related to a particular process, the user has a way to set Server attributes outside the Server definition. This is done using the following syntax:

```
setEnv <serverName> <attributeName> <attributeValue>;
```

The <attributeName> can be maxLoad, maxMemory, maxSwap or mountpoint, like inside the server.

Finally, the Server has an " ***attach*** *<coordName>; "* directive, which attaches the given Coordinator to the Server (see section IV.1.1 for a reminder of the Attached Coordinator concept), and a *"**detach** <coordName>;"* directive.

The definitions of FileAgent, FolderAgent and ProcessAgent share a few common attributes, among which is the mandatory "server" attribute. It states which Server the Agent refers to. This attribute will not be mentioned anymore in the following explanations.

The FileAgent definition starts with the keyword "***FileAgent***" and can have the following attributes:

- filename (M): A string. The full path of the file.
- owner: A string. The owner of the file.
- grp: A string. The group to which the file is associated.
- permissions: Three octal digit. The Linux permissions of the file.
- attributes: A string. The Linux attributes of the file.
- md5: the checksum of the file.
- content: Any POSIX compliant regular expression that the file content should match. This attribute can be used multiple times, in which case the file has to match all the patterns.

The FolderAgent definition starts with the keyword "***FolderAgent***", and has the same attributes as the FileAgent, except for the md5. It has however two attributes that the FileAgent does not have:

- nameFilter: restricts the content that will be analyzed based on the regular expression given as argument (*e.g.*: *.txt for all the text files, log-2013-* for the log files from 2013 only, *etc*)
- maxDepth: the maximum depth at which the folder and its sub-folders will be explored.

The ProcessAgent definition starts with "***ProcessAgent***", and has the following attributes:

- procName (M): A string. The name of the process, with the full path to the binary and the arguments, as seen in the linux ps command.
- user: A string. The user running the process.
- maxCpu: A percentage. The maximum CPU consumption the process can have. The default value is 100.

- maxMemory: A percentage. The maximum memory consumption the process can have. The default value is 100.
- command: the command to execute to start the process.
- service: the name of the Linux service the process is associated with.
- parent: A string. The name of the ProcessAgent representing the parent process. The child ProcessAgent will inherit all the attributes from the parent ProcessAgent.
- multiplicity: an integer. Denotes how many processes matching this agent can be running on the server. The default value is unlimited (-1).

Although the *"command"* and *"service"* attributes are not mandatory, it is suggested to give at least one of them. If none is provided, the procName will be used as a starting command.

There is one attribute for each of the process' limits that the system can set. The name of the attribute is the name of the limit (as written in the /proc/<pid>/limits file), replacing the space by an "_", and removing the preceding word *"max"*. For example, the limit name *"Max cpu time"* has the attribute *"cpu_time"*. For a full list and a full definition of the limits, please refer to the linux kernel manpage.

In the definition of the FileAgent, FolderAgent and ProcessAgent, the user can define *"Recovery Rules"*, or *"Triggers"*, as defined in section IV.1.1. A Trigger definition is as follows:

```
<when> <action1> trigger <action2> on <target>;
```

<when> can either be "***before***" or "***after***". The <action1> and <action2> are specific actions that are taken either on the Agent being defined (<action1>) or on the <target> Agent (<action2>). The list of actions will be detailed later in this document.

The definition of Coordinator is different from the definition of Agents. Like the Agents, it starts with the keyword "***Coordinator***", followed by its name, and the definition between curly braces. The differences lie inside the braces. There is no attribute as before, but rather a list of directives.

First of all, as a Coordinator is meant to aggregate other MetaAgents, there is a directive which is used to express what the Coordinator is supposed to aggregate. This is done with the "***needs***" directive. It has the following syntax:

```
needs <MetaAgentType> <needName>;
```

where the MetaAgentType is one of "***Coordinator***", "***FileAgent***", "***FolderAgent***" or "***ProcessAgent***", and needName is just a name to uniquely identify this need.

A need then requires to be assigned. This is done with the following syntax:

```
<needName> = <MetaAgentName>;
```

where needName is the name of the need as defined previously, and MetaAgentName is the name of the MetaAgent that fulfill this need.

Since it can be handy to define and assign a need at the same time, the following syntax is supported:

```
needs <MetaAgentType> <needName> = <MetaAgentName>;
```

The Coordinator definition also contains the *"Dependency Rule"* definitions. Such a dependency is defined with the following syntax:

```
requires <CoordinatorName>;
```

where CoordinatorName is the name of the Coordinator from which the current Coordinator is dependent.

To associate a Veto to the Coordinator:

```
veto <vetoName>;
```

It sometimes happens that not all the children are needed to work fine so that the full service works. For example, if the work is distributed between ten servers, it is maybe not a critical condition if a couple of them die. This is the purpose of the "***tolerate***" directive. It is written the following way:

```
tolerate <n>;
```

which means that the service represented by the Coordinator will be considered okay up to n children having a problem. This can be very dangerous and has to be used with great care. The way it works internally is that the Coordinator will check its children in the order given by the reinforcement algorithm. If one returns a faulty behavior, but the threshold set by the "***tolerate***" directive is not reached, it is ignored and the diagnosis continues. Depending on the order in which the children are checked, one might have unexpected behavior. This feature is recommended only in Coordinators where all the children are strictly the same and redundant.

The last directive in a Coordinator is the following:

```
set <varName> <value>;
```

This gives a value to a given variable, where the value can be another variable name.

The values given to the attributes in the Agent definitions can be variables. Variables start with "$" and do not need to be previously declared. The interest of the variables is to factorize the information. For example, the definition of an application with processes, files and folders, all running on the same server. Instead of hardcoding the name of the server in all the Agent definitions, it is set to a variable, and the actual server name is assigned to the variable in the top Coordinator. If one ever moves the application to another server, the Server name has to be changed in one place only. When assigning a value to a variable, all the children of the Coordinator assign the same value to the variable.

Variables are most relevant in the inheritance process. The inheritance relation is specified when declaring an entity by appending the following:

```
1 : <parentName>
```

The parentName is the name of the MetaAgent from which the current MetaAgent inherits, very much like in C++ programming. The complete definition of a MetaAgent inheriting from another one is as follows:

```
1 <MetaAgentType> <childName>: <parentName>
```

Of course, the child and parent MetaAgent have to be of the same type.

When the inheritance mechanism takes place, much information is copied from the parent to the child. This includes all the attributes, information regarding the Shared Experience mechanism, the *"needs"* in case of a Coordinator, and the unassigned variables.

As a consequence, one could fully define a standard installation of a standard application (for example Apache, MySQL), using variables for the unknown parameters (such as the user running it or the md5 of the configuration file). Then every instance of that application can be fully defined in a few lines. It is enough to create a Coordinator inheriting from the top Coordinator of the abstract application, in which the values of the variables are set.

Because it is often more convenient to split the configuration into several files, an "***import***" directive was defined, which is as follows:

```
1 import <filename>;
```

These directives must be at the beginning of a file, and the file extension must be *".phr"*.

Appendix B contains a more formal description of the grammar, as well as many examples covering the various aspects we have seen.

IV.2.1.iii The compiler

Now that we have seen how a user configures the software, it is interesting to understand how the configuration is read and interpreted. This section will give details about the algorithms used to parse the user input, as well as some implementation technicalities.

The parser gets the configuration file and folder paths either from command line arguments or from an option file. The very first task is to sort these files and make sure that there is no cyclic dependencies, based on the import statements. This is done using Tarjan's algorithm [Tarjan 1971]. This Algorithm is a graph theory algorithm for finding the strongly connected components of a graph. A directed graph is called strongly connected if there is a path from each vertex in the graph to all other vertices. This means that if a graph where the vertices are the files and the edges are the dependency requirements is defined, there is no circular dependency in the files if and only if there is no strongly connected component in the graph using Tarjan's algorithm.

Note that the files are sorted in the correct order, but the MetaAgent definitions within a file have to be ordered by the user.
If a circular dependency is found, it is signaled to the user and the program exits. Otherwise, a sorted list of files is obtained.

From a global perspective, the following functions are in turn applied to all the files from the configuration list:

- Server parsing
- ProcessAgent parsing
- FileAgent parsing
- FolderAgent parsing
- Coordinator Parsing
- setEnv constraints parsing

Each of the MetaAgent types and the Servers are associated to a class that holds their configuration in memory. The purpose of the functions listed above is to create all the object instances from the flat configuration files.

Core At the center of the process sits a class called Core, which acts as a big repository for all the MetaAgents and Servers created. Among other things, it provides ways to register new instances, ways to retrieve the MetaAgent objects by their associated names and it keeps a list of the instances currently being treated

(there is one type of each entity in this list).

Server parsing The first step is the Server parsing. When a Server definition is encountered, a new Server object is created, as well as a new EnvironmentAgent object and a FileAgent object to describe the fstab file of the server. All three instances are registered, and set as currently being treated. Every attribute which is then parsed is passed to the current Server object. If the attribute is the server address, the Server object keeps it for itself. Otherwise, it forwards it to the EnvironmentAgent associated with it. The EnvironmentAgent will assign the parsed attribute and value to its associated object attribute. If the attribute is *"mountpoint"*, it adds a *"content"* property (see section IV.1.1 FileAgent for a reminder) to the FileAgent representing the fstab file instead. When an *"attach"* directive is encountered, the name of the required Coordinator is appended to a list, that will be used later. The *"detach"* directive removes a name from this list.

Process parsing Following the same principle, the Process parsing function will loop through all files. When it encounters a ProcessAgent definition, it creates the instance, registers it, sets it as the currently active ProcessAgent, and then set the attributes depending on the read properties. If the property being read is the *"parent"* property, the Core is asked to return a link to the parent object. This link is given to the child, and all the attributes copied from the parent to the child object. If a ProcessAgent has a parent ProcessAgent, it always has to be declared before any other property.

File parsing For the FileAgent parsing the function will create a new object, register it and set it as currently active when parsing a FileAgent definition. The only exception to the classical *"property read sets one attribute in the object"* rule is the *"content"* property, because it may appear several times. The way this is handled is that the FileAgent object contains a list in which all the *"content"* properties are appended.

Folder parsing The FolderAgent parsing is very similar to the FileAgent parsing, since the FolderAgent class inherits from the FileAgent class.

Trigger parsing A treatment which is common to all Agents is the Trigger definition. Each Agent object has a list of Trigger objects associated with it. Contrary to the rest of the classes, the Trigger class only contains the name of the MetaAgent to which it refers (as opposed to the MetaAgent classes which contains

reference to the object). This has to do with the inheritance mechanism we will see later in this section.

Coordinator parsing The parsing of the Coordinators is slightly more complicated, since it is not properties but directives that are parsed. The only exception is the *"tolerate"* directive, which can be represented as an attribute. Regarding the *"needs"* definitions and assignments, this is done using internal dictionaries:

- The first one, called the *definitionDic*, is populated when reading a need definition. The key is the *"needs name"*, and the value is the type of MetaAgent expected.
- The second one, called the *assignmentDic*, is populated when reading a *"needs assignment"*. When reading the assignment, the Core is asked to return the object associated with the MetaAgent name to be associates. If its type corresponds to the one expected in the definitionDic, an entry is added in the assignmentDic, with the need name as the key, and the value a link to the MetaAgent object to which it refers. Otherwise the error is signaled and the program exits.

The *"Dependency Rules"* are managed with a simple list. When a *"requires"* property is read, the Core is asked to return the object associated with the name and it is pushed to that list.

The last function to be executed is the *"setEnv"* parsing. When such a directive is read, the Server object is loaded from the Core using the Server's name, and the properties in the directive are given to it.

At the end of the execution of all these functions, the Core contains all the objects described in the configuration files. There are two important aspects whose internal behavior have not yet been described: variables and inheritance handling.

Every MetaAgent has a dictionary called *"variableDic"*. The keys are the names of the variables (starting with "$"). The values are the lists of attributes to which the variables are assigned. For example, a FileAgent is defined the following way:

```
1 FileAgent myFile {
      filename -> $var_filename;
3     server -> $srv;
      owner -> $var_name;
5     grp -> $var_name;
  }
```

It will have the following entries in the variableDic:

- $srv: [server]

- $var_name: [owner, grp]
- $var_filename: [filename]

This dictionary is one of the attributes which is passed to the child ProcessAgent, as well as to the inheritance mechanism.

When a variable is assigned a value using the "***set***" keyword in a MetaAgent, the MetaAgent looks into the dictionary to know which attribute the variable is assigned to. It then sets the value to all the attributes in the dictionary. In the Coordinator, the assignment mechanism is then passed recursively to all children. In order to be properly propagated to all the Children of the Coordinator, the *"set"* directives have to be written at the end of the Coordinator definition.

Variables can be used anywhere where the properties are treated as a normal class attribute, which means everywhere in the Agents except for the FileAgent content. Also, the value assigned to a variable can be another variable.

The inheritance mechanism takes place when a new MetaAgent is defined. When MetaAgent A inherits from MetaAgent B, a deep copy of B is executed first. What this means is that all the dictionaries, all the needs, but also all the children in case of Coordinator inheritance are actually copied: a copy of the whole tree is done. To keep all names unique, the names of the newly created children have the original name, with a suffix being the name of the class being defined.

Fig. IV.3 illustrates this principle. All of the *"A tree"* is defined first, and then states that the Coordinator B inherits from A. The *"A tree"* is copied, and all the children keep their original names to which *"_ B"* is appended. To avoid excessively long names in case of chained inheritance, the original name are only concatenated with the latest inheritance. If a Coordinator C was inheriting from B, the children would be named *"A_ x_ C"*, *"A_ y_ C"* and so on.

The next task is to attach the Coordinators to the Servers as required, using the list kept by the Server and mentioned earlier: for each name in the list, the associated instance is retrieved from the Core, and pushed to another list in the Server.

The following step consists of assigning a so-called *"classification"* to every Coordinator. The classification uniquely identifies a Coordinator and its position in a tree. The classification is composed of a succession of integers. The first integer is the tree index, and the next are the edge indices necessary to reach the Coordinator. On the example of Fig. IV.4, possible classifications would be:

- E: 1
- F: 1;1

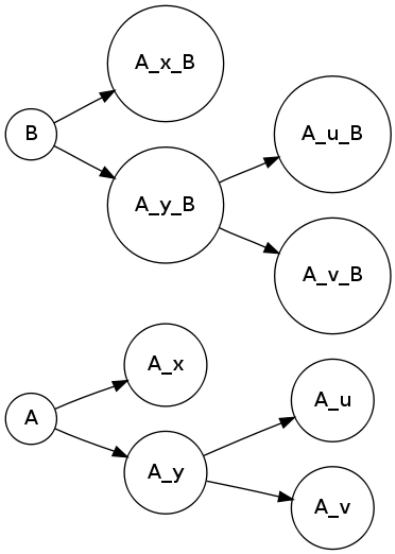


Figure IV.3: Automatic naming of MetaAgent created by inheritance

- A: 2
- B: 2;1
- D: 2;1;1
- C: 2;2

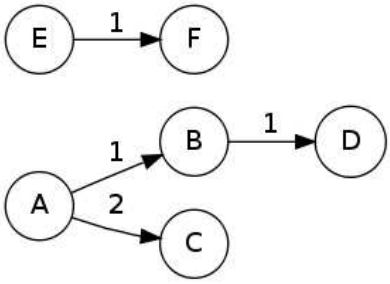


Figure IV.4: Example of Classification

This classification is used in the Core software that we will describe later.

Note that so far, no mention was done regarding the reinforcement learning concerns at the compilation step. The reason is that it requires to understand the algorithms involved, and that they will be described later only. For now one can

consider that what relates to them can be described as a standard class attribute, and are thus copied and shared via inheritance.

Once all the configuration is parsed, it is necessary to distinguish the Abstract entities from the others. An entity is considered Abstract if it is not fully defined, that means if there are still unknown parameters regarding it. An Agent is fully defined if all the mandatory fields described in section IV.2.1.ii are completed, and if no variable remains unassigned. For a Coordinator, being fully defined means to have all the variables assigned, all the needs assigned, and its parent as well as children fully defined. Abstract entities are later ignored by the Core software, but they hold a crucial role in the Shared Experience mechanism.

The last step is to load this configuration into the database. Because the database cannot be changed while the Core software is running, the parser actually produces an SQL script. The user has to decide by himself when to load it into the database.

The straight-forward way of doing this would be to completely erase the database, and recreate the configuration in it. While easy to apply, this method has a major drawback: all the experience that was gained thanks to the reinforcement learning would be lost. The solution is to load the content of the database into memory, compare it with the configuration files and commit only the changes to the database. However, because it might sometimes be desired, the user has the possibility to apply the straight-forward method: in this case, the database is dropped, all the tables are recreated, and all the objects are simply inserted in the appropriate order.

The algorithm used to modify the database without losing its previous content is more complex, and consists of a graph matching like algorithm. There is one consideration which is very important to keep in mind: a Coordinator and more generally an entity is considered as changed if and only if the tree under it has changed. This mean the following:

- An Agent will be considered the same, even if its attributes (for example the filename or the process user) have changed. The reason of this choice is because we prefer to consider the Agents *"logically"*. For example, the Agent used to represent a configuration file of a web server is not defined by its attributes (filename: /etc/httpd/conf.d/mySite.conf; owner: root; grp: root; and others) but by what it represents. It is a file containing the configuration for a website. A change in the website configuration will change the configuration file, but not what it represents.

- A Coordinator is considered changed if any of its children, direct or indirect in the hierarchy was changed.

- Coordinators whose parent changed are not necessarily considered modified. Changing A_x in Fig. IV.3 will change A but A_y, A_u and A_v will be considered unchanged. The experience gathered by them is still valid. This is useful in situations when a graph is split. For example, a Coordinator WebSites aggregates two Coordinators, SiteA and SiteB. Later, the configuration is split, the Coordinator WebSites is removed, and SiteA and SiteB become independent. The experience gathered by the two sites independently still remains valid. That is why a change higher in the tree does not necessarily propagate down. While a change in the parent does not change the children, it does modify the links between them.

At a high level, the algorithm consists of the following steps:

- Load all the MetaAgents and the MetaAgent tree which are in the database, as well as the dependency rules and the Servers.
- Based on the name, establish lists of new, deleted and unchanged Servers/MetaAgents.
- The deleted entities will be removed at the beginning of the SQL script.
- For the unchanged entities, the indices used internally are renumbered, so that they match those in the database. The SQL script then uses *"update"* directives.
- The new entities are assigned ids which do not conflict with those used in the db, and are inserted by the script.

Appendix C contains an illustration of the tree comparison algorithm.

The actual implementation of the parser is done using Python and the library pyparsing. Pyparsing is a library which contains many facilities to describe a file content structure as a set of syntactical rules.

There are two reasons for using Python (as opposed to the rest of the software):

- The main reason is that the introspection mechanisms, the weak typing, as well as other dynamic features make the designing of a parser much easier than with C++.
- While the C++ part of the software has quite a few dependencies that we will see later, the compiler can work with just a few python files. Thus, this portability allows users to easily test configuration on their local machines.

IV.2.2 Remote Agent

The remote agent is a piece of software that runs on all the machines the user wants to supervise. Its only purpose is to answer queries from the Core. As such,

it makes it a fairly simple program. Wait for a query, perform the task, send back the answer. The complexities of it are more at the technical level, and are just implementation details. The query concerns all the attributes of files, folders, environment or processes.

The agent is developed in C++, using several boost libraries [Boost-team 2013].

The choice of C++ was made to keep the same programming language as the core, but also for the tight coupling between C++ and the system calls. Because of constraints on the boost::serialize version and in order to reduce the dependencies of our software, it is statically compiled against boost.

The queries are made over the network using a C++ class called *"Request"*. This class has three main attributes:

- Type: what type of request it is
- Target: what the request refers to
- Require: what does the user want to know

The Type attribute is one of the following:

- Alive: this request is the simplest one, and just expects an empty answer. The Target and Require arguments are empty. The goal of this request is to make sure that the remote agent is running and is responsive.
- Environment: this request concerns information about the environment of the server. The Target is empty in this case as well.
- PidTree: this request asks for the full process id tree based on a process command line. The Target contains the command line to look for.
- ProcessPid: this requests information about a process given its PID, which is set in Target.
- File: requests information about a File given in Target.
- Mountpoint: given a filename in Target, this requests information about the mountpoint on which it is stored.
- Folder: a request for the content of a folder given as Target

Sometimes the user does not want all the information regarding an entity. For example, if the software does not have anything to compare it with, it might be useless to ask for the checksum of a file. This is the role of the Require attribute: only the information explicitly stated there will be fetched.

Intuitively, the Environment requests are made by EnvironmentAgents, the Folder requests are made by FolderAgents, and so forth. The Mountpoint request are done by FileAgents, FolderAgents and EnvironmentAgents.

While there is a unique class to make the query, there is as many answer classes as query Types. The attributes of those classes are the same as those of the class they represent: the ProcessPid answer class will have similar attributes as the ProcessAgent class, and so on.

There is one extra attribute shared by all the answer classes, which contains errors that might have been encountered while fetching the information required. The error is then handled appropriately by the Core.

While very basic, this approach of only answering what is needed without further manipulating the data offers a nice decoupling between the algorithm logic and the information collection. If the algorithms of the Core change, it does not impact the agent running on the clients, as long as the informations that are required are not different.

Finally, there is a certain amount of parameters the user can tune, such as the timeout duration to execute the query, or the port on which to listen for the Core's queries. Note that all the Remote Agents must listen on the same port.

IV.2.3 Interaction with the user

There are two kinds of interaction between the software and the user:

- Output: it is important that the user knows what the software is doing, or what it has done.
- Input: as stated before, the software is not monitoring anything, it expects to be notified about problems by either a user, a physical person, or any third party program.

This bidirectional communication is made possible using an Application Programing Interface (API). The purpose of the API is to provide the user with an access to the software functionalities. This access should be as agnostic as possible to the internal complexity.

The API, called phrApi, is based on the Observer design pattern, visible in Fig. IV.5. It handles the outgoing communication from the software to the user.

The principle is that the client, called an Observer, does not actively check if there is new information available from the Subject or Observable. Instead, the

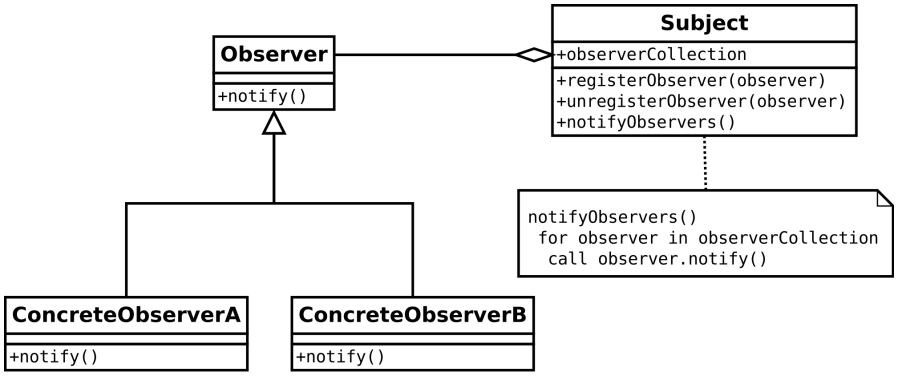


Figure IV.5: The Observer Design Pattern

Subject will notify the Observer any time there is something new.
The great advantage of this approach is that it implies a much lower load on both the Observer's and the Observable's sides, compared to active polling.

The input communication is similar to a Remote Procedure Call (RPC): the Core is notified that the user wants a given action with given arguments to be executed.

All the communication is done over the network using similar technical implementations as for the Remote Agent. The messages are exchanged using a single class called *"api_ message"*. It has four main fields:

- Type: the type of message
- Text: the content of the message
- Date: the date when the message was issued
- Time: the time when the message was issued

The Date and Time are only used for the output communication. The Text attribute contains either what the Core has to report, or the parameters the Client gives to the Core. The Type attribute can be one of the following:

- newMsg (output): a *"Message"* is information which is sent to the user and which requires an acknowledgment. They are used to make sure the user sees the message. Of course, it is up to the interface implementation to keep it displayed until it is acknowledged; the Core will keep track of it until it is.
- ackMsg (input): acknowledgment for a given Message.
- newLog (output): informative text.

- newQuestion (output): the Core sometimes requires input from the user. It is done using this type.
- newProblemQuestion (output): a specialization of the newQuestion Type. The Core requires the user to tell it what the current problems are.
- newYnQuestion (output): another specialization of the newQuestion Type. It represents a *"yes or no question"*.
- answerQuestion (input): sent by the Client to answer a question
- currentProblem (output): when the Core is treating problems, it notifies all the observers about it.
- giveProblem (input): a facility to use answerQuestion, which is used to specifically notify the Core about ongoing problems.
- setVeto (input): used to set the value of a Veto.
- newVeto (output): used by the Core to notify clients that a Veto was set.
- statistic (output): used by the core to send the value of counters in order to make statistics.
- error (output): this is used to notify the Client about a problem in the communication. For example, if the answer to a *"yes or no question"* is *"maybe"*.

The phrApi abstracts all the complexity of the input communication by providing simple methods:

- *bool isConnected()*: return true if the Client is connected to the Core, false otherwise
- *void answerQuestion(int, std::string)*: answer to a given question, identified by an index
- *void ackMessage(int)*: acknowledge a given message, identified by an index
- *void giveProblemList(std::string)*: notify problems to the Core
- *void setVeto(std::string,bool)*: set a value to a veto

For the output communication the reaction to a given message is completely dependent on the Client. The user has to specify what to do. This is done by implementing *"virtual void update (api_ message)"*, which is the callback method for a message received by a Client. Appendix D contains examples of implementations of usable Clients, based on the phrApi.

Based on the phrApi, several ready-to-use user interfaces were developed:

- phrUtils: a command line tool for all the functionalities listed above

- phrGUI: still being prototyped. A GUI interface based on the Qt framework.
- phrXml: only for the output communication. This stores all the output into an XML ring buffer file.
- phrSimu: an interface used by our simulation software describe later.
- phrIcinga: an interface that gathers data from Icinga, the monitoring software used at LHCb.
- phrWeb: a web interface based on phrXml.

The phrApi does not give access to other information that a user might want, such as the list of Servers or MetaAgents, the state of the reinforcement learning algorithms, the history and so on. The reason is that the Core does not provide such functionalities. Some information still has to be fetched from the database.

The most convenient way to display information to the user wherever they are is with a web interface: phrWeb.
Based on the Python Framework *"Django"*, phrWeb exposes all the content of the database to the user, and allows the modification of some of the data. Since the database does not contain *"live information"*, such as problems being treated, or logs, it was necessary to make use of the phrApi. This is the point of the phrXml interface: by writing all the output into a formatted XML file, its content can be dynamically loaded into the web page using AJAX technology.

Note that there are two special observers: the *"cout"* and the *"syslog"* observers. Those are threads of the Core, and are used to print to the console of the Core (when not daemonized) and to use the syslog facility.

IV.3 Core

The Core part of the software is the central piece which contains all the algorithms used to actually diagnose problems and offer recovery solutions. In a first part, we will describe the different aspects the Core has to deal with, and which algorithms are used to address them. The second part will contain more technical details regarding the actual implementation.

IV.3.1 Algorithms

Throughout the previous sections, we have mentioned several aspects of Phronesis that were left aside until now, for example the reinforcement learning aspects or the diagnosis approach. This section will now cover all those algorithms in depth.

IV.3.1.i Sorting algorithm

When several problems occur at the same time the question with which one to start arises. In section III.2, we have seen that a human expert addresses the issue by *"sorting"* the problems based on their respective dependencies, as well as the likelihood of each to be responsible. This sorting is made possible by the knowledge the expert has about the system as well as the experience he accumulated.The filtering that the expert is doing on the information he receives from the user or the monitoring system is very important as well.

To reduce the noise (filtering) of the input, the Core uses the following criterion. If several problems concern the same Coordinator tree, only the deepest reported Coordinators are considered. The reason for this is quite straight forward: the root problems are always to be found in the leafs of the trees. The lower a Coordinator is in the tree, the closer it is to the potential real problem.

To illustrate it, the example of several websites, Site1 and Site2, aggregated under the Coordinator Web can be used again. Depending on how the monitoring is implemented, an actual failure in Site1 could trigger alarms concerning Site1 and Web. If Web was considered in the diagnosis, the wrong branch of Site2 might be explored. If it is considered *"normal"* to have a problem on Web, because one of its children, namely Site1, has a problem, then it can be ignored. Only Site1 would be left to diagnose, which is what is expected.

The question is then how to find which one is the lowest in the tree. Doing a graph exploration for every reported problem is not efficient at all because of the cost of the operation. Moreover, this information is completely static. One can compute it once, and store the result in the database. This is the role of the *"Classification"* attribute of the Coordinator seen previously. Thanks to this, filtering the problem list is done with simple integer comparison.

After the noise is filtered out, the entities need to be analyzed in the appropriate order. It would be useless to start analyzing the problem on a website if the software was also informed about a problem on the storage where the site gets its files from. That is where the dependency rules start to matter.

Every Coordinator is associated to a *"Weight"*, which represents how important in a system a given Coordinator is. The higher the Weight, the more Coordinators depends on it.

The formula is the following:

$$W(X) = 1 + \sum_{n} W(X_n) \tag{IV.1}$$

This means that the weight of a Coordinator X is 1 added to the sum of the weights of all the Coordinators that depend on X.

It could be objected that by doing this, an *"absolute"* Weight is calculated, which is not based only on the problems currently being treated. Dependencies that are not notified as problematic are also taken into account. While slightly more time consuming, computing an absolute Weight presents two advantages:

- First of all, and very important, the sorting result would be exactly the same than with a Weight taking into account current problems only.
- A Coordinator with a high absolute Weight represents a central service. Starting the diagnosis on the highest Weight Coordinator can pay off, even if none of the other currents problems depends on it. Even if known dependency rules say otherwise, there might be unknown dependencies not yet in the database.

Also, while not completely static because of the dependency rules that can change dynamically, the absolute Weight of a Coordinator can be somehow cached, and thus reduce the computation time of the Weight calculation.

Once all the problematic Coordinator Weights are calculated, they are sorted in descending order.

If several Coordinators tied for the highest Weight, the one that had the most problems in the past is chosen.

If there is still a draw between several Coordinators, a random one is picked.

This step tells in which order to analyze the problems given as input. The actual diagnosis algorithm will be detailed later in this chapter (see section IV.3.1.v).

IV.3.1.ii Recovery Algorithm

In this part, we will explain the recovery steps that follow the discovery of the root cause of a problem. As already mentioned in the section III.2, once the root cause is found, correcting the mistake is normally quite easy. What is less trivial is how to have the correction be taken into account.

First let's look at the possible root causes, as well as the potential solutions.

The tables IV.1, IV.2 and IV.3 will list the problems one can encounter with various entities, the solutions to them, as well as a name for the situation. These names are the action names to be used when defining the Triggers in the configuration.

Name	Problem	Solution
FILE_MOUNT	The device where the file is supposed to be is not mounted	Mount the device
FILE_CREATE	The file is not present	Create it
FILE_CHOWN	The file has the wrong owner	Assign it the right owner
FILE_CHGRP	The file has the wrong group	Assign it the right group
FILE_CHMOD	The file has the wrong permissions	Assign the correct permissions
FILE_CHATTR	The file has the wrong attributes	Set the correct attributes
FILE_PATTERN	Not all the patterns match the file	Correct the file content
FILE_MD5	The file does not have the correct checksum	Correct the file content

Table IV.1: Problems and solutions for File problems

Name	Problem	Solution
PROCESS_START	The process is not running	Start it
PROCESS_RESTART	The process is not behaving as it should (too high CPU, mem...)	Restart it
PROCESS_STOP		Stop the process
PROCESS_ULIMIT	Some of the system limits of the process are not properly set	Set them properly
PROCESS_CHUSER	The process is not running under the right user	Restart it with the proper one
SERVICE_START	The process (associated with a linux service) is not running	Start it
SERVICE_RESTART	The process (associated with a linux service) is not behaving properly	Restart it
SERVICE_STOP		Stop the service
SERVICE_RELOAD		Reload the service

Table IV.2: Problems and solutions for Process problems

Name	Problem	Solution
ENV_MOUNT	A mountpoint is not mounted	mount it
ENV_UNMOUNT		Unmount a mountpoint
ENV_DISK	A device is full	Free space
ENV_INODE	No more free inode	Free some
ENV_OVERLOAD	The system is overloaded	Free some resources
ENV_ON	The server is off	Turn it on

Table IV.3: Problems and solutions for Environment problems

Those actions cover most of the situations one can encounter. Note that a keyword *"ANY"* exists, that the compiler will understand as *"all the possible actions"* for the type of MetaAgent being treated. The Trigger rules defined by the user define the consequences that those actions will have. For example, after changing the group of the httpd configuration file, restart the httpd process. Building the full recovery procedure is not easy. In fact, a restart of the httpd process might require steps to be executed before or after.

The complexity of this task comes from the fact that two graphs are needed to describe all of the Trigger rules: one for the *"After"* rules, and one for the *"Before"* rules. One has to bear in mind that the Trigger rules are not symmetric: *"after actionX on A, do actionY on B"* does not imply *"before actionY on B, do actionX on A"*, and hence the complexity. To address this problem, we found two algorithms that satisfy our needs.

The first algorithm is a custom solution designed to answer our particular needs. The basic idea is to construct a single *"trigger graph"* from the *"before graph"* and the *"after graph"*, where every edge is a requirement, whatever the direction. When the software needs to find the full recovery procedure starting from an atomic action A, it first finds the connex components of A in the *"trigger graph"*. This connex components are all the actions that will be involved one way or another. The second step is to sort those actions. For this, the following comparison criteria is used:
For two actions A and B, $A < B$ if one of the following statement is satisfied:

- There exists a path from A to B in the *"after tree"*
- There exists a path from B to A in the *"before tree"*

One can compute the complexity of this algorithm in the worst case. We define:

- $|Vt|$ = cardinality of vertices of the Trigger graph
- $|Et|$ = cardinality of edges of the Trigger graph
- $|Va|$ = cardinality of vertices of the After graph

- $|Ea|$ = cardinality of edges of the After graph
- $|Vb|$ = cardinality of vertices of the Before graph
- $|Eb|$ = cardinality of edges of the Before graph
- $|N|$ = cardinality of the connex components of the start element

By definition $|Vt| \leq |Va| + |Vb|$ and $|Et| \leq |Ea| + |Eb|$
All the graphs are constructed only once at the startup of the program, therefore their construction cost is not counted here.
The first step is to find the connected part of the start element in the *"trigger graph"*, which can be done with a Breadth First Search algorithm (BFS). Its complexity in the worst case is $\Theta(|V| + |E|)$. In our case, we have $\Theta(|Vt| + |Et|)$. The BFS returns a list of vertices of cardinality N that need to be sorted.
The used sort algorithm performs in the order of $N \cdot log(N)$ comparisons, where N is the amount of elements to sort.
Our comparison criteria consists of checking the existence of a path in two different graphs. This check can be done using again a BFS algorithm. Thus, in the worst case, when comparing two elements, the complexity is $\Theta(|Va| + |Ea| + |Vb| + |Eb|)$.

In the end, the complexity of the algorithm is:

$$\Theta(|Vt| + |Et| + N \cdot log(N) \cdot (|Va| + |Ea| + |Vb| + |Eb|)))$$

Using the inequality of $|Vt|$ and $|Et|$ and defining $|Vm| = max(|Va|, |Vb|)$ and $|Em| = max(|Ea|, |Eb|)$:

$$\Theta(|Vt| + |Et| + N \cdot log(N) \cdot (|Vm| + |Em|))$$

In practice, many layers of caching and optimizations were added that dramatically reduce the complexity. In particular, the results of comparisons are stored, so in the long term the complexity is rather of the order:

$$\Theta(|Vt| + |Et| + N \cdot log(N)) \tag{IV.2}$$

The alternative algorithm is to build a new graph every time a full recovery procedure will be needed. The new graph represents the execution sequence of the actions. It is made by starting from the start action in the *"after"* and *"before"* graph, and then explore those graphs. While the edges from the *"after"* graph are simply copied, the edges from the *"before"* graph are inverted. An edge *"A -> B"* in the *"before"* graph means that *"B has to be executed before A"*, so in the new graph one wants a *"B -> A"* edge.

In order to build the new graph, as many BFS as vertices in the new graph are needed in the worst case. Using the same notation as before, building the new graph has a worst case complexity of:

$$\Theta(N \cdot (|Vm| + |Em|))$$

The second step of the algorithm is to do a topological sort on the new tree, which has a complexity of

$$\Theta(|Vt| + |Et|)$$

In the end, the total worst case complexity of the second algorithm is

$$\Theta(N \cdot (|Vm| + |Em|) + |Vt| + |Et|) \tag{IV.3}$$

Given the similarities in the worst case complexity of the two algorithms, and the fact that we do not have any control over the graph type, both of them were implemented, and the user can make his choice in the configuration file.

IV.3.1.iii Reinforcement Learning Algorithm

The role of the reinforcement learning algorithm is to allow performance improvements over time and to take advantage of previous experiments. In our case, this translates into faster and better diagnosis.

Drawing again a parallel with section III.2, the human expert speeds up his diagnosis by recalling previously encountered situations of the same or a similar problem. For a given problem, the expert will check his hypothesis based on the likelihood of each of them. The likelihood of the hypothesis is evaluated based on previous experiments, and every repetition will modify the probability. It is the very same behavior we wish to reproduce with our software.

Ultimately, the goal can be stated like this: *"given a problem, find the real root cause as fast as possible"*, or in other words *"starting from a problematic Coordinator, find the child Agent responsible for it as fast as possible"*. Note that here one considers a single Coordinator, and that one tries to improve the exploration of its tree. Not improving the choice between several Coordinators reported by the user. This is treated in the following section.

In our case, the optimal speed is reached when the tree is traversed directly using the correct path that leads from the faulty Coordinator to the responsible leaf agent. In the worst case, a complete traversal of the tree is made; in the best case, the correct path is taken directly. The challenge is to optimize this choice based on previous experience.

There are a great variety of algorithms treating the subject of machine learning. While some are classified as *"unsupervised learning"*, those that interest us are under

the branch *"supervised learning"*. This means that the algorithm gets feedback regarding the output it produces (for a complete review, see [Duda 2000]). In our context, there is a direct feedback. It is the comparison the Agent will do between the values it expects and the values it measures. If they match, it means that this agent is not faulty, and that the choice was not good. On the other hand, if it finds a mismatch in the agent, it has chosen the appropriate path, which means that it has to keep this information, and *"reinforce"* this path for future uses.

The choice of reinforcement learning algorithms is quite large: Q-learning, TD-learning, *etc*. Some strong constraints have to be kept in mind:

- All those algorithms have tunable parameters, that are adjusted either based on user's preferences or at the learning phase. For example the learning rate or the discount factor of the Q-learning algorithm. The problem is that the optimal values for these parameters are strongly dependent on the situation, and that there is no such thing as an absolute good value. This contradicts the requirement for a generic software. On top of it, no cross-validation is possible.

- Because we cannot afford to have a long learning phase, the Shared Experience must be used. This means that the algorithm itself has to be compatible with the principle of sharing and extrapolating experience.

The algorithm that we finally decided to use is very simple but very efficient: every edge in a Coordinator tree is associated to a counter. When a path turns out to be correct, the counters from all the edges that make up this path are incremented. This is a trivial way of knowing which decisions were good.

Regarding the Shared Experience, the manipulation is straight forward as well. The counters are shared, as illustrated in Fig. IV.6. The labels on the edges indicate the counter IDs which are used. One can see that the common part of the Coordinator tree shares the same counter ID, whereas the other edges have different counter IDs. When one of the sites increases the value associated to the counter IDs, all the other sites gain this experience.
Note that all parts of the path are updated. If F1 happens to be faulty, the values of the counters 1 and 4 will be incremented. What this means is that even if the other Sites do not know anything about F1, they gain the experience that there was a problem with the configuration of the Site. Once again, this follows the approach of considering the MetaAgents logically, that is for what they represent.

The counter associated to an edge is called *"Occurrence"*. While not strictly necessary for the reinforcement algorithm, each Coordinator has also a counter associated which is called *"Total"*, and which represents the total amount of time a Coordinator was diagnosed as faulty. As for the Occurrence, the Total counter is shared between similar Coordinator Trees. Because it might be of interest to the

user, there are local Occurrence and Total for each node and edge, which are never shared.

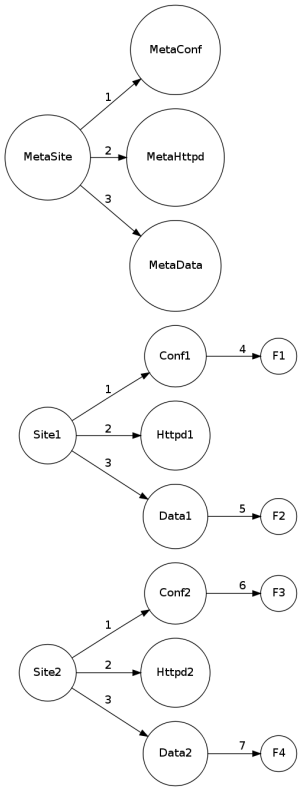


Figure IV.6: Illustration of the Shared Experience mechanism

In section IV.2.1.iii, when describing the compiler algorithms, we explained that the data relevant to the reinforcement learning can be considered as standard class attributes. These attributes are the Occurrence and Total IDs.

Finally, there is another great advantage of using such a simple system which is not to be neglected. One of our goals was to come up with an algorithm whose output should be corrected by the user in case of wrong diagnosis. This algorithm answers our need better than expected since the user can not only correct a wrong diagnosis by changing the counters, but he can also give a priori knowledge to the algorithm without any learning phase.

The methodology previously described allows to gradually construct a probability tree. The question then arises on how to traverse it. There are two strategies that were considered:

- Explore the children in the order of decreasing probability. The advantage of this approach is that one will go directly to the child which is the most often culprit. On the other hand, if the responsible is the least probable, one will explore the entire tree before finding it.

- Make a weighted random choice between the children, where the weight is the probability measured on previous experience. While less performant than the previous method in the best case, this approach is better for the worst case.

In order to choose one of the methods, we decided on a criterion which is the average amount of attempts to find the responsible child, given the probabilities of the tree.

To find this number, one can consider that the tree has only one level, because the Occurrence of a given level is the sum of the Occurrences of the lower level. The following can then be defined:

- $N(X)$: average amount of attempts to find out that X is responsible

- η: average amount of attempts to find the responsible

- V_x: Occurrence of X

- $V_{tot} = \sum_k Vk$: sum of the Occurrences of all the Agents of the tree

From the previous definitions, the probability that the agent X is responsible is given by $\frac{V_x}{V_{tot}}$

First policy We write $\{X_1, X_2, ..., X_n\}$ the Agents ordered by decreasing occurrence. To find the responsible Agent, one needs, by definition:

- 1 attempt with a probability of $\frac{V_{x1}}{V_{tot}}$

- 2 attempts with a probability of $\frac{V_{x2}}{V_{tot}}$

- ...

- n attempts with a probability of $\frac{V_{xn}}{V_{tot}}$

So for the first policy:

$$\eta = \sum_{i=1}^{n} i \cdot \frac{V_{xi}}{V_{tot}} \tag{IV.4}$$

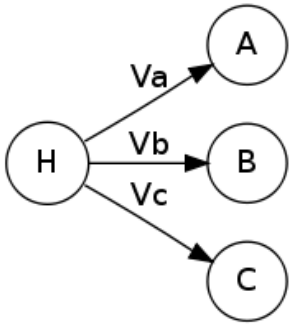


Figure IV.7: Example to illustrate the second policy

Second policy To understand how to compute η for the second policy, one can take a simple example with only three branches.

To calculate N(A) in Fig. IV.7, which is the average amount of attempts one needs to choose A when A is the responsible, one needs to know the probability of choosing A in the first (N1(A)), second (N2(A)) and third (N3(A)) position. By following the probability tree:

$$N1(A) = \frac{V_a}{V_a+V_b+V_c} = \frac{V_a}{V_{tot}}$$

$$N2(A) = \frac{V_b}{V_a+V_b+V_c} \cdot \frac{V_a}{V_a+V_c} + \frac{V_c}{V_a+V_b+V_c} \cdot \frac{V_a}{V_a+V_b} = \frac{V_b}{V_{tot}} \cdot \frac{V_a}{V_{tot}-V_b} + \frac{V_c}{V_{tot}} \cdot \frac{V_a}{V_{tot}-V_c}$$

$$N3(A) = \frac{V_b}{V_{tot}} \cdot \frac{V_c}{V_{tot}-V_b} \cdot \frac{V_a}{V_{tot}-V_b-V_c} + \frac{V_c}{V_{tot}} \cdot \frac{V_b}{V_{tot}-V_c} \cdot \frac{V_a}{V_{tot}-V_c-V_b}$$

We get: $N(A) = N1(A) + 2 \cdot N2(A) + 3 \cdot N3(A)$

Knowing the average amount of attempts for each entity, and the probability that a given entity is faulty, one can write:

$$\eta = \frac{V_a}{V_{tot}} \cdot N(A) + \frac{V_b}{V_{tot}} \cdot N(B) + \frac{V_c}{V_{tot}} \cdot N(C)$$

The formula can be generalized. We denote with $\{X_1, X_2, ..., X_n\}$ the Agents, without any specific order. If, for an element X, we assign Ω_k^X as all the arrangements of size k, with X in the last position, we can write:

$$N(X) = \sum_{i=1}^{n} \left(i \cdot \sum_{\Omega_i^X} \prod_{k=1}^{|\Omega_i^X|} \frac{V_k}{V_{tot} - \sum_{j=1}^{k-1} V_j} \right) \tag{IV.5}$$

The j being an index for the element in the set, and not a counter as such.
Finally:

$$\eta = \sum_{i=1}^{n} N(X_i) \cdot \frac{V_i}{V_{tot}} = \sum_{u=1}^{n} \frac{V_u}{V_{tot}} \cdot \left(\sum_{i=1}^{n} \left(i \cdot \sum_{\Omega_i^X} \prod_{k=1}^{|\Omega_i^X|} \frac{V_k}{V_{tot} - \sum_{j=1}^{k-1} V_j} \right) \right) \quad \text{(IV.6)}$$

It is impossible to say if one or the other policy is better. It all depends on the case. Neither is there an easy criteria to define which one to use. Simulations of the two policies are shown in section V.1.3. For this reason, we decided to let the user choose the algorithm he thinks is the best for his environment.

IV.3.1.iv Dependency algorithm

One of the most interesting features of our software is that it is able to find dependencies between Coordinator trees based on previous experience. In a way, it is also a reinforcement learning. This capacity allows our software to infer new Dependency rules, and thus provides better diagnoses.

The importance of the dependency rules should not be neglected. Not only are they helpful to make a faster diagnosis, but without them, some diagnoses would simply be wrong or the recovery impossible. Let's illustrate this with the following example: suppose that one has a simple web server, which gets its content from an NFS server. If the NFS server becomes unavailable, people would experience problems on the website. If one declares the two problems, NFS and Site, and if the dependency rule is not known, there is a certain chance to start diagnosing Site, and eventually find a problem. While the problem is real, there is no way for our software to know that this problem is just an artifact of another problem. Even worse: there is absolutely no action that could be taken on the Site to correct the problem, so one would constantly loop. Of course, there is a protection against this situation. However, if one starts with the NFS problem, the issue of the website will either resolve itself or at least be solvable afterward.

This is why the dependency rules are so crucial. Because the dependencies are sometimes not as obvious as the previous example, or sometimes unknown, having the software discovering them for the user is a great advantage.

The new rules are discovered using the user or the monitoring software feedback. When problems are flagged, the software keeps the list of problems in memory, and then tries to diagnose one. Once the diagnostic and the recovery solution is calculated, the user is prompted to give an update on the situation. The user then presents a new list of problems. It is the difference between two consecutive lists that allows to see what was the impact of fixing the entity that was diagnosed earlier. In the previous example, if the website problem disappears after diagnosing NFS, the dependency *"Site depends on NFS"* can be inferred.
All the problems that are not present anymore at the next iteration create a

new dependency rule. If the rule already exists, an occurrence counter on this rule is incremented. The reason for having such a counter is that the rules are not active by default. This means that they are not taken into account by the software. It might happen that the monitoring was flickering for a reason or another, and this would then result in a new rule, which happens to be wrong. To avoid this, the user has to manually approve the rule, and the counter is a good indicator of how *"reliable"* the rule is. For convenience, the user has the option to set a threshold after which the rule will be automatically validated.

While the principle is easy to understand, the actual application of it is much more complex. There are indeed many cases that are not as obvious, and require careful thinking. For example, can new rules be inferred if the diagnosed problem is still reported as faulty, but other problems are gone? There is no definite answer to this, it all depends on the situation. The problems related to a Server, with their EnvironmentAgent or their attached Coordinators, also require special treatment. All those cases make the implementation of this dependency discovery mechanism a fairly complicated piece of logic.

Another consideration is what to do with new problems that appear after a diagnosis was made. It is not possible to know whether they are a consequence of our previous recovery attempts, or if they are independent. Even if they were a direct consequence of our action, there is nothing much one could do. The user should maybe define more Triggers if this is what is required. Completely ignoring the new problems would not be reasonable either. The consensus found is to keep a detailed history of previous situations and use it at the proper time, but without impacting the logic. When a diagnosis has been made and the suggested action has been applied by the user, the action taken, the list of problems before and after and a user comment are stored in the database. The next time a problem occurs on the same Agent, the user will be warned with all this history.

The process of deducing dependencies between entities exists in the literature as *"Association rule learning"*. It is a form of data mining which focuses on discovering relations between variables in large databases. Rakesh Agrawal was a pioneer in the domain with his Apriori algorithm, and he has contributed in many important papers such as [Agrawal 1993], [Hipp 1998], [Srikant 1995], [Srikant 1996a], [Zaki 1997] or [Pasquier 1999]. Those algorithms however focus on transactions and do not have any notion of order in the items.
"Sequence mining" answers the need of finding sequential patterns, once again initiated by Agrawal in [Agrawal 1995]. It is a very active field of research (for example [Mannila 1997], [Srikant 1996b]) and is nowadays applied a lot to track users' Internet browsing or shopping habits (see [Chen 2009], [Pei 2000], [El-Sayed 2004], [Liu 2007] or [Zheng 2004] for some examples).
The situation of Phronesis is a bit different, because the rules are not searched for in a large database, but inferred step by step. This avoids the complexity

required by all of the algorithms mentioned above to cope with the big amount of data. However, the dependency algorithm uses set intersections, and so do many algorithms, like the famous *"ECLAT"* [Zaki 2000].

IV.3.1.v Diagnosis algorithm

We have now seen all the different pieces that makes each operation possible. This section will explain how all these different algorithms work together to achieve the final goal.

Everything starts when an ongoing list of problems is given to the software. Each problem is first associated to its Coordinator. The list of Coordinators is then filtered based on their depth in the tree, and sorted based on the Dependency rules, as explained in section IV.3.1.i.

The Coordinator trees are then traversed in turn, until a faulty one is found. The exploration is made using the algorithm of section IV.3.1.iii. If a Coordinator tree is visited without finding a problem, this Coordinator is added to an *"Undiagnosed list"*. The interest of this list is to avoid to recheck a tree that was already found to be sane.

If a faulty Agent is found, the recovery procedure describe in section IV.3.1.ii is triggered. Note that this does not execute any action, but prompts to the user the exact steps he should follow to solve the problem.

The problem list is then updated, following section IV.3.1.iv. The whole process is started again until the Problem list as well as the Undiagnosed list are empty.

While this global picture works in most cases, there are two cases that require special handling.

The first case concerns the EnvironmentAgent as well as the Attached Coordinators, which are normally not reported to the software. When exploring a normal Coordinator tree, the attributes of files, folders and processes have to be verified, using their respective Agents. The measurements are done by contacting the Remote Agent described in section IV.2.2. Before verifying any attribute, all those Agents will verify that the remote Server is pingable, and that the Remote Agent is running properly. If any of these tests fail, the problem is considered to come from the server, and it is a so-called *"Environment problem"*. It might also happen that even if the server and the remote agent are up and running, the problem comes from the environment (heavy load, disk problem, *etc*). If this is the case, it would only be noticed after going through all the reported Coordinators, and none of them being faulty. In such a situation, a complete environment check

(EnvironmentAgent and Attached Coordinators) of the Servers is done until a faulty one is found.
Once the failed server is found, all the problematic Coordinators that have an Agent running on the faulty server are reported to the user as being affected. The standard recovery procedure then applies the EnvironmentAgent or the Attached Coordinator.

The second case happens when no problem could be found in the reported Coordinators, and all the environments were found to be sound. In that case the Coordinators that the reported problems depend on are loaded, based on the valid Dependency rules. The full diagnosis process starts again with this new batch of Coordinators. If again no problem is found, the non valid Dependency rules are used, and the process starts all over again. If even then no problem can be found, the software gives up.

Note that in case of Environment problems, or when the software works on non reported Coordinators, no new dependency rules are inferred.

IV.3.2 Implementation

The Core software was designed in C++, making use of several boost libraries. The choice of C++ was made to ensure the best performance. At the beginning of the development, it was not clear what the CPU and memory footprint would be, nor which computing power would be needed. We decided to go for a safe choice, and use C++, despite all the overhead of complexity compared to a language like Python.

The design is object oriented and is multi-threaded.

There are two main threads, which are for the interactions with the user, and for the Core functionalities. A few more threads might be started on demand for specific cases, mainly related to the Remote Agent queries or for user interactions.

UML diagrams are given in appendix E regarding the detailed implementation, but some highlights are given here. First of all, the code is split in three parts:

- `phrDbApi`: this part is compiled as a dynamic library and provides an interface between the Core and the database.
- `phrLib`: this dynamic library contains the elementary classes that are used by the Core.
- Core: this package contains mainly the algorithms that make use of the `phrDbApi` and the `phrLib` to provide the diagnosis and recovery solutions.

The phrDbApi is a unique class which contains all the helper methods one needs to query the database, such as *"retreiveAllCoordinators", "incrementOccurrence", etc.*

In phrLib, the equivalent to the classes described in the parser section is found: Server, MetaAgent, Coordinator, Agent, FileAgent, FolderAgent, ProcessAgent and EnvironmentAgent. The classes AgentRecovery, FileRecovery, ProcessRecovery and EnvironmentRecovery, given a faulty Agent, will propose the necessary recovery action.

Several classes are used for communication purposes, either with the user or the remote agents: Client, Message and Connection, UserMsg, UserQuestion, and the like.

The Core contains mainly the following classes:

- SyncManager: used for thread synchronization
- ServerPool: aggregates and manages all the Server objects
- MetaAgentPool: aggregates and manages all the MetaAgent objects and objects inherited from it
- InteractionManager: manages the interaction with the user
- ProbabilityManager: manages all the counters related to probability and reinforcement learning
- RuleManager: manages the Dependency Rules as well as the triggers
- VetoManager: deals with the list of Veto and their values
- Aggregator: manages the lists of ongoing problems, filters the problem list, takes care of the comparison of successive problems list, triggers the recovery procedure and manages history.
- Scheduler: sort the problem list according to dependencies, start the diagnosis of a Coordinator Tree
- Recovery engine: constructs all the recovery procedures of a faulty agent,

The global picture of the implementation of the diagnosis algorithm is shown in Fig. IV.8:

To start it, the software needs to be provided with a mandatory configuration file which will be detailed later. With no other option, the process will start, stay in the foreground, and wait for user's input. There are however different running modes. The *"daemon"* mode runs just as the default one, that is it waits for the

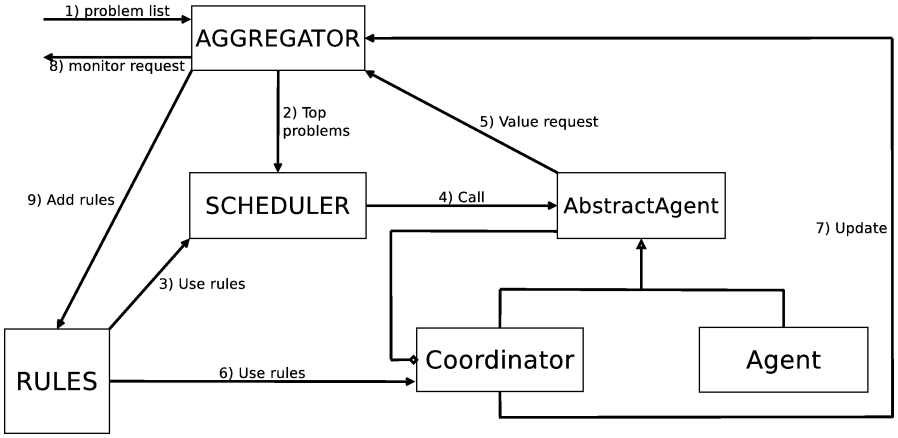


Figure IV.8: Implementation principle of the diagnosis algorithm

user's input, but runs as a daemon process in the background. Those two modes never exit, unless demanded by the user.
The *"list"* mode allows the user to start the program with a list of problematic Coordinators. Once all Coordinators are diagnosed, the program exits.
The *"full"* mode will check every single Agent listed in the database, report on them, and exit.
The *"checkAll"* mode will make the diagnosis of all the Coordinators in the database. This is very different from the *"full"* mode. While the *"full"* mode will ignore all kinds of dependencies and relations between Agents to make raw tests, the*"checkAll"* mode will proceed with all diagnosis steps (filtering, dependency, *etc*) as if all the Coordinators were reported as problematic. At the end, it reports, and exits.
Note that the *"full"* and *"checkAll"* modes do not give recovery solutions.

The configuration file needed to start the software contains the following information:

- Database information: server address, port, user, password, name needed to connect to the database.
- Interaction information: the port number for the user to connect and interact.
- Remote Agent information: the port number to connect to the Remote Agents (note that all the Remote Agents have to be configured to listen on this same port); the timeout after which the software considers the Agent unresponsive (this has to be tuned taking into account the execution timeout on the Remote Agent side)
- Syslog information: the user can activate logging using the Syslog facility, and tune the amount and type of information he wants to be logged

- Cout information: when running in the default mode, information is printed on the console. The user can tune the amount and type of information he wants to be printed
- Algorithm parameters: options such as the automatic validation threshold for dependency rules, the choice of graph traversal algorithms, or the random seed.

At the time of writing, we are still consolidating the implementation, fixing bugs and adding extra capabilities to the software. Doxygen is used to ensure a proper code documentation, and the code is checked against the static analysis of cppcheck.

Chapter V

Results

Contents

As Lhcb runs a large farm, that cannot be disturbed for the sake of our tests, it was important to be able to simulate the behavior of the software in various situations. To achieve this goal, a tool was developed which takes care of simulating problems given a probability distribution, interacts with the Core just as a user would, and produces some statistics. The first part of this chapter shows the different simulations we ran. The second part details the current utilization of our software in our production environment.

V.1 Simulations

It was important in order to test our algorithms to be able to simulate realistic situations. To achieve this result, we developed a complete set of tools to produce Monte-Carlo simulations.

Phronesis needs to be compiled in a special way. The reason is that the simulation tool tests the algorithms of the Core, and not the code quality of the Agents: when in a normal usage, an Agent would query the remote server to get an information and process it, in simulation mode, the Agent is instructed what to return. This allows to simulate any kind of environment on one single machine.

Another piece of software is used to randomly generate problems based on user's input, inject signals to the Core to fake the agent's analysis, interact with it to confirm or deny its diagnoses, and produce statistics about the behavior of Phronesis. This tool allows to reproduce almost any kind of environment, including the LHCb one.

V.1.1 Importance of the dependency rules

The idea of the first simulation was to present to the software a use case which is typical of what can be found in classical — and in the LHCb — environment, and see how it behaves. The focus was on the dependency rules.

The test bench is the following:

- One NFS server
- One MySQL server
- One website (Site1) getting its content from MySQL
- Another website (Site2) getting its content from NFS and MySQL

The configuration entered in the database is the following:

- A Coordinator "Nfs", which contains three FileAgents for the export file and the two files it serves, and one ProcessAgent for the NFS processes.
- A Coordinator "Mysql" which contains only a ProcessAgent for the mysqld process
- A Coordinator "Site1" which contains one FileAgent for the file it serves and two ProcessAgent for the httpd processes
- A Coordinator "Site2" structured as "Site1". Those two inherits from the same template in order to share their experience.
- A Coordinator "Web" which contains the Coordinators "Site1", "Site2" and "Mysql"

To make the experiment more interesting, the dependencies between those various entities are not known by the software at the beginning. The test protocol consists of creating up to four simultaneous random problems on the various systems and measure the actions of the software under various circumstances:

- No dependency rule used (valid0 in Fig. V.1, Fig. V.2 and Fig. V.3).
- Dependency rule valid after appearing 200 times (valid200 in Fig. V.1, Fig. V.2 and Fig. V.3).
- Dependency rule valid after appearing 100 times (valid100 in Fig. V.1, Fig. V.2 and Fig. V.3).
- Dependency rule valid as soon as appearing (valid1 in Fig. V.1, Fig. V.2 and Fig. V.3).

To add even some more realism, the problems are reported as they would be seen by a classical monitoring system:

- A real problem on the NFS server will trigger alarms on Nfs, Site2 and Web
- A real problem on Site1 or Site2 will trigger alarm on Web and the associated Site
- A real problem on Mysql will trigger an alarm on Site1, Site2, Web and Mysql

The first positive outcome is that in every situation, all the dependency rules were found by the software.

The first value of interest is the percentage of diagnosed problems in the various tests. Fig. V.1 makes clear that the situation improves a lot: from 93.2% to 99.9%. An analysis of the simulations shows that the non spotted errors are actually found, but considered as false positives.This situation appears when distinct problems happen at the same time and have similar consequences on the alarms sent by the monitoring. For example, if the nfs or the http daemons are down, Site 2 becomes unreachable for the monitoring. If both daemons are down at the same time, the NFS daemon needs to be fixed first.If the software decides to fix httpd first — because there is no priority rule — it will get the false positive situation.

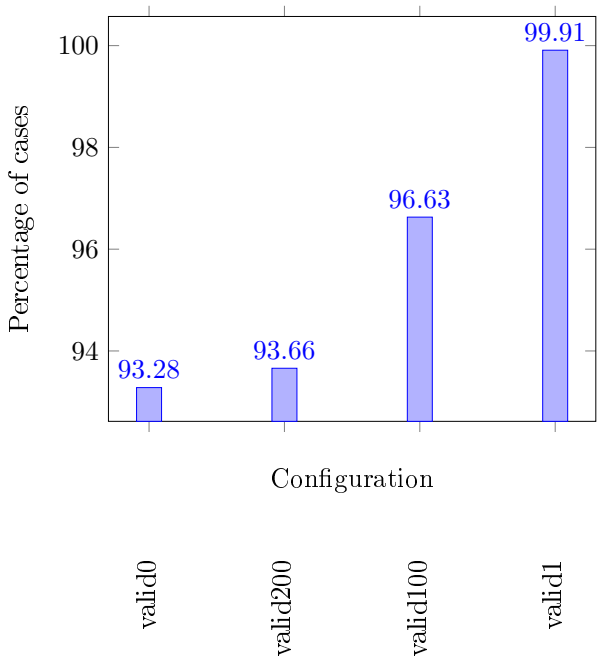


Figure V.1: Percentage of cases with correct diagnoses (Simu 1)

The second value of interest is how many time a users or a monitoring tool needs to interfere with the software in order to solve a problem. In order to update

probabilities and deduce priority rules, the aggregator needs to update its alarm list after each attempt to solve one of the problems. In case of n concurrent problems, at least n interactions are needed. Fig. V.2 shows the percentage of extra actions (i.e. more than the optimal) in different situations. The priority rules clearly help to improve the situation. The 40% of the cases that required additional user intervention, when no rules were used, are managed in an optimal way if the rules are used as soon as they are deduced.

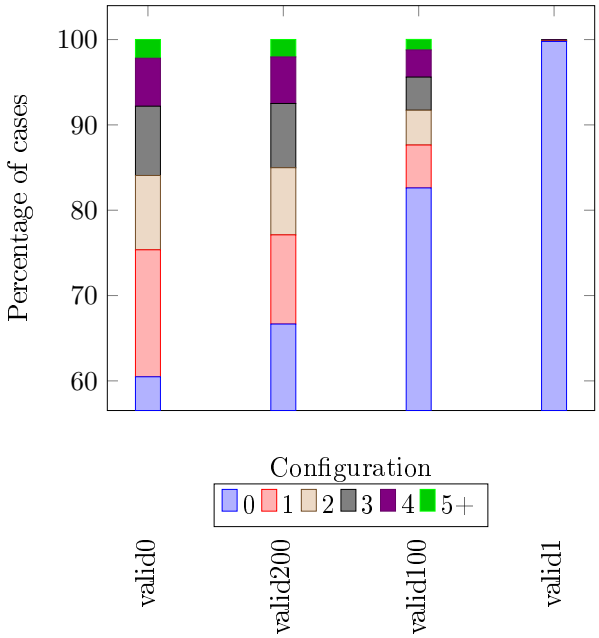


Figure V.2: Additional user actions with respect to the optimal solution (Simu 1)

To finish, it is interesting to compare the amount of involved agents when solving problems. Less involved agents, means faster solving. Fig. V.3, similar to Fig. V.2, shows how many additional agents are visited. In that case as well, the priority rules improve the percentage of optimal resolutions from 57.63% to 95.95%. The percentage of situations requiring more than one additional check decreases from 36.05% to less than 1%.

V.1.2 Importance of the Shared Experience

As seen previously, the Shared Experience principle reduces the configuration workload for the user, but also shows very useful in environment where many replications cannot be ran to train the reinforcement learning algorithms. The second simulation attempts to demonstrate this.

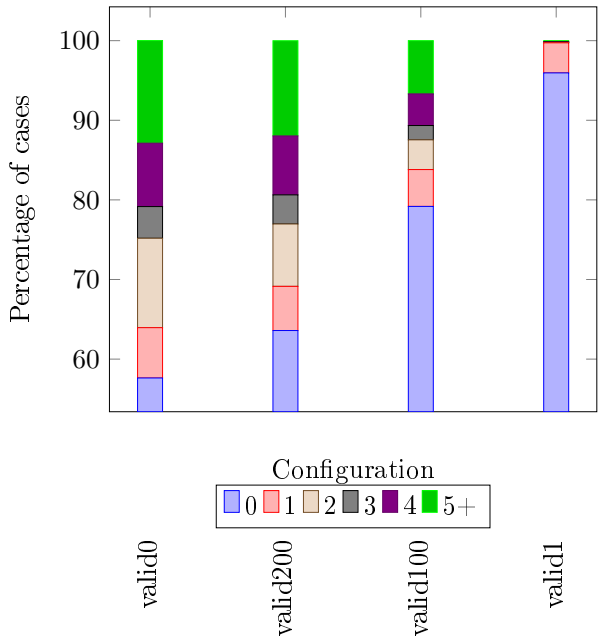


Figure V.3: Additional visited agents with respect to the optimal solution (Simu 1)

The test case simply consists of 20 Coordinators, each of them having 20 FileAgents, on which problems were simulated. In 62% of all cases a selected file had a problem. The other 38% of failure are equally shared between the remaining 19 files. Fig. V.4 is a comparison of the amount of visited Agents after only 20 replications in both cases. While the Coordinators using the Shared Experience offer an optimal resolution in 57.14% of cases, the independent Coordinators reach only 28.57% of cases.

Fig. V.5 and Fig. V.6 illustrates the learning speeds by comparing the number of optimal resolutions : while the *"shared"* curve reaches the optimal value (62% $\pm$ 5%) after about 70 replications, the *"not shared"* curve needs about 400 replications.

This situation of having perfectly equivalent Coordinators is of course idealistic. However, considering that the greatest share of software has a deterministic behavior, it is safe to consider that several instances of the same Software running on different machines will have the same behavior, and thus the same failures.

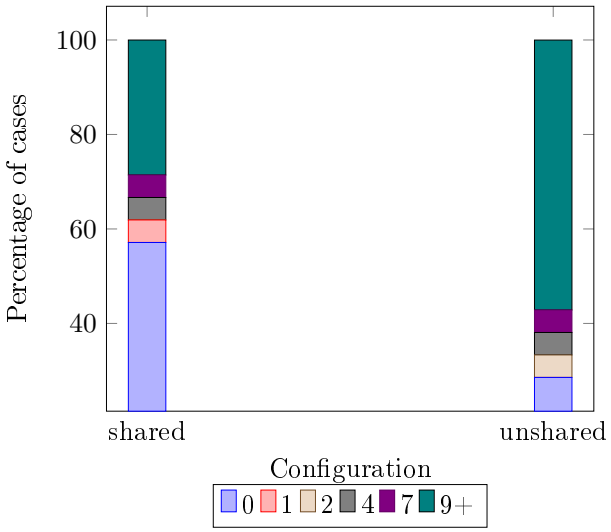


Figure V.4: Additional visited agents with respect to the optimal solution after 20 replications (Simu 2)

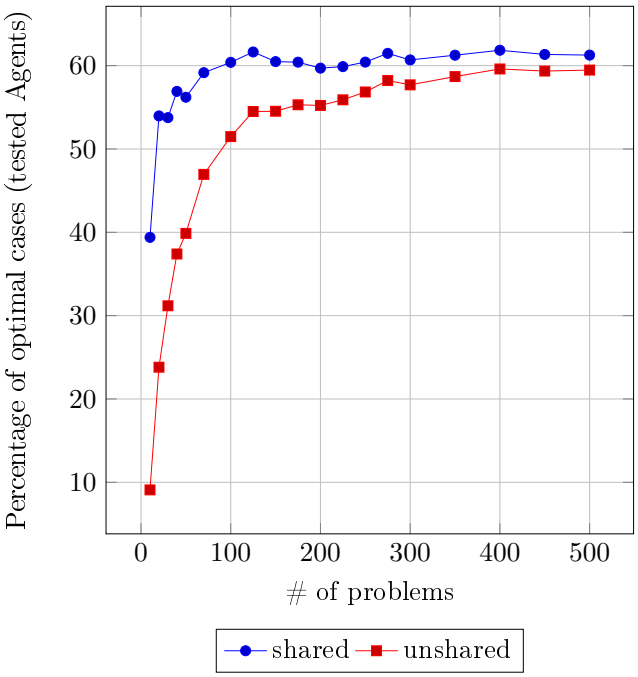


Figure V.5: Comparison of learning speeds (Simu 2)

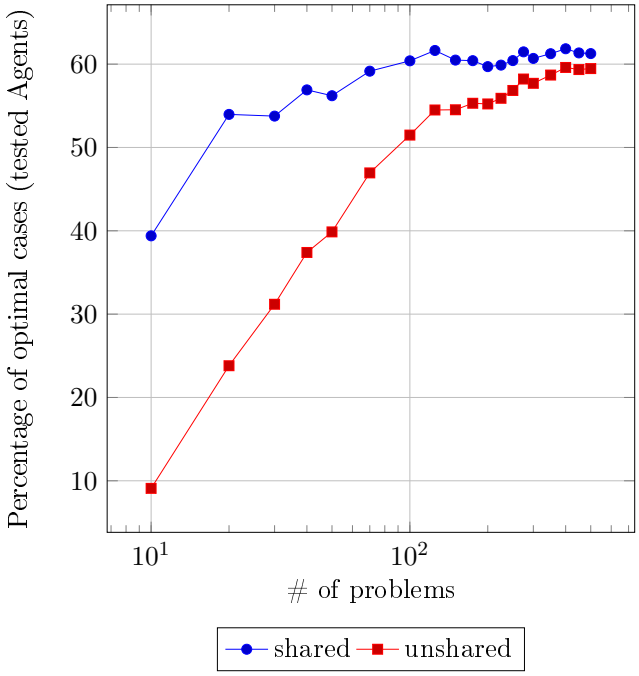


Figure V.6: Comparison of learning speeds using logarithmic scale (Simu 2)

V.1.3 Importance of the priority algorithm

We have discussed in section IV.3.1.iii the two strategies to choose the order in which the MetaAgents are analyzed. The mathematical formulas that give an average amount of visited agents are too complex to be compared as such. In this section, we will look at simulations covering a wide range of situations, and see how the two algorithms, called *"best"* and *"rnd"* perform.

The test protocol consists of one Coordinator with ten Agents, on which the created problems follow various probability distributions. 200 repetitions were made. The probability distributions represents the following situations:

- All Agents have an even probability of being faulty (scenario *"even"*)
- Half the Agents have a higher probability than the other half (scenario *"1/2"*)
- A quarter of the Agents have a higher probability than the other three quarters (scenario *"1/4"*)
- The probability difference between two "consecutive" Agents increase by a factor k, where k is in 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9 (scenario *"k..."*)
- One Agent has a much higher probability than the others (scenario *"1+"*)

- One Agent has a much lower probability than the others (scenario *"1-"*)

First of all, Fig. V.7 shows that in all the situations, the average amount of visited Agents is almost identical : the algorithm using weighted probability (*"rnd"* algorithm) needs in average 0.06% (with a 1.97% standard deviation) more attempts to find the good one that the algorithm using sorted probabilities.

A more detailed analysis (visible on Fig. V.8, Fig. V.9, Fig. V.10, Fig. V.11, Fig. V.12, Fig. V.13, Fig. V.14, Fig. V.15, Fig. V.16, Fig. V.17, Fig. V.18, Fig. V.19, Fig. V.20, Fig. V.21 and Table V.1) of each situation reveals that, as expected, the weighted probability curve is more equalized than the other one.

All even Fig. V.8 shows the average amount of attempts per node when the probabilities of each node are the same.

Half dominant Fig. V.9 shows the average amount of attempts per node when the probabilities of half of the Nodes is much higher than the other half.

k=1.1 Fig. V.10 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.1

k=1.2 Fig. V.11 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.2

k=1.3 Fig. V.12 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.3

k=1.4 Fig. V.13 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.4

k=1.5 Fig. V.14 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.5

k=1.6 Fig. V.15 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.6

k=1.7 Fig. V.16 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.7

k=1.8 Fig. V.17 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.8

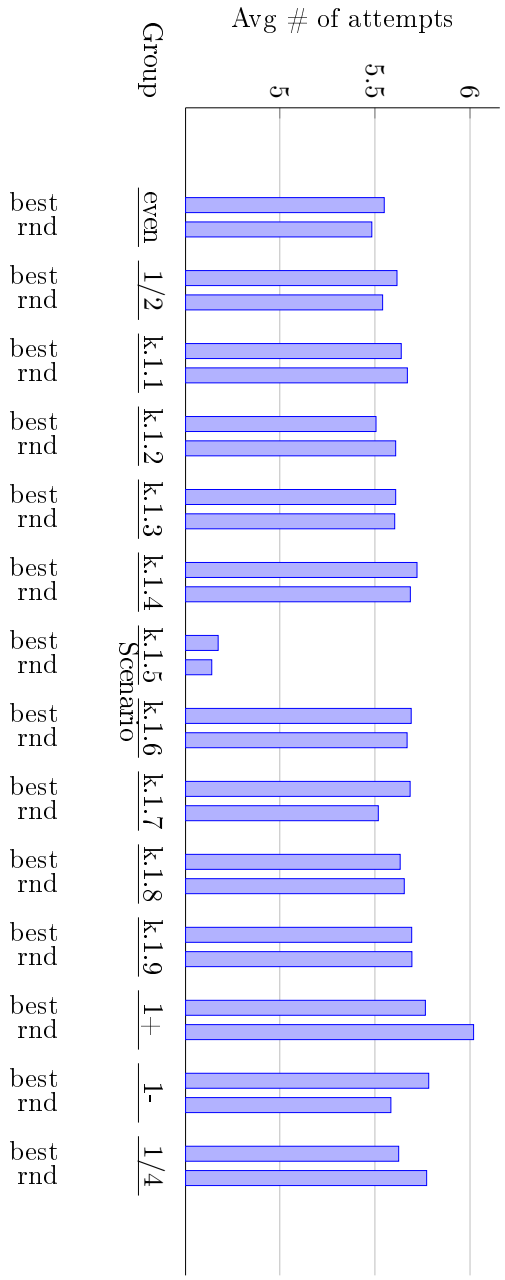


Figure V.7: Avg # of attempts to solve a problem

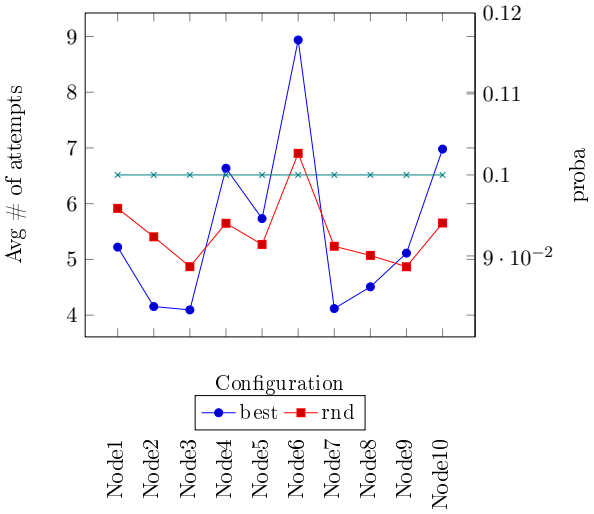


Figure V.8: Comparison of the avg # of attempts per Node in scenario *"even"*

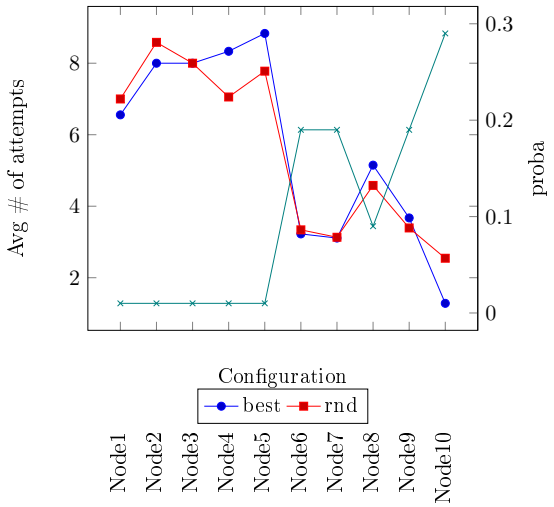


Figure V.9: Comparison of the avg # of attempts per Node in scenario *"1/2"*

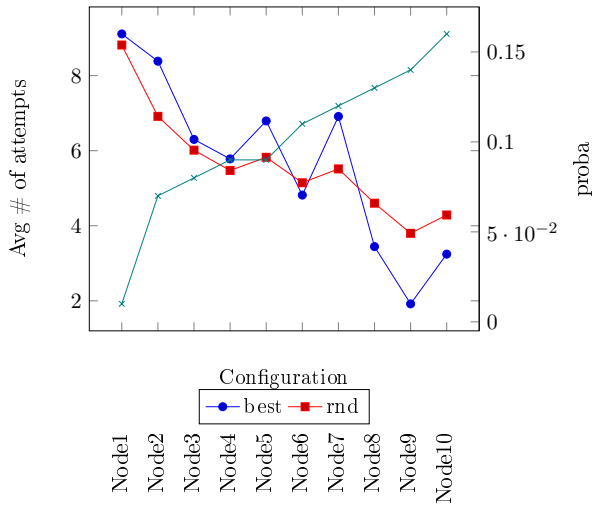


Figure V.10: Comparison of the avg # of attempts per Node in scenario *"k=1.1"*

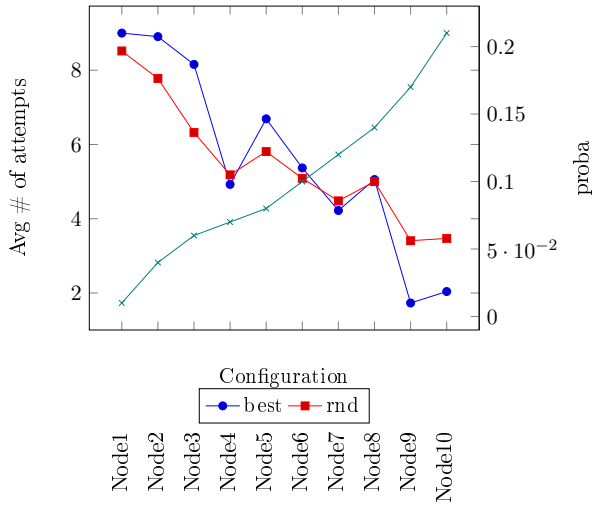


Figure V.11: Comparison of the avg # of attempts per Node in scenario *"k=1.2"*

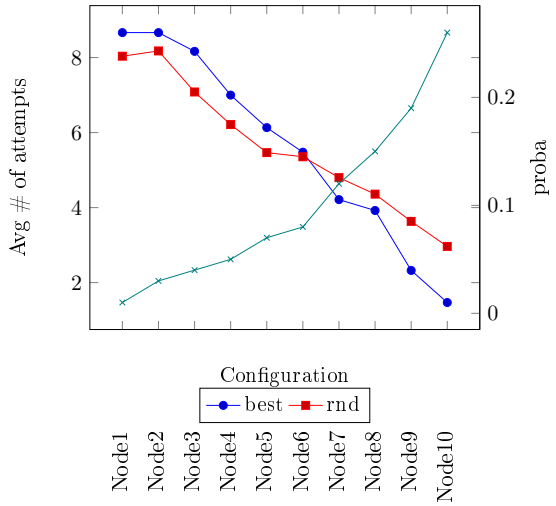


Figure V.12: Comparison of the avg # of attempts per Node in scenario *"k=1.3"*

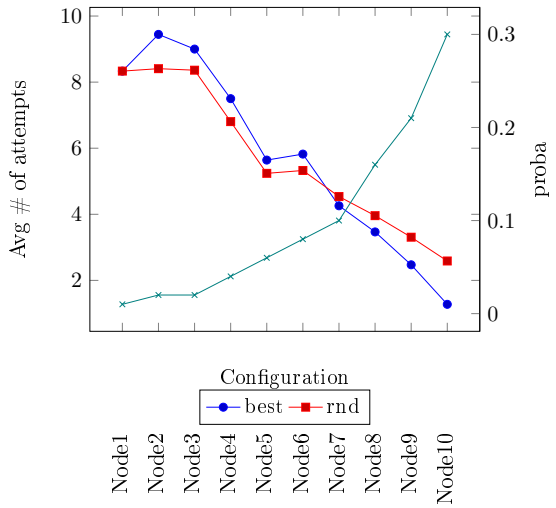


Figure V.13: Comparison of the avg # of attempts per Node in scenario *"k=1.4"*

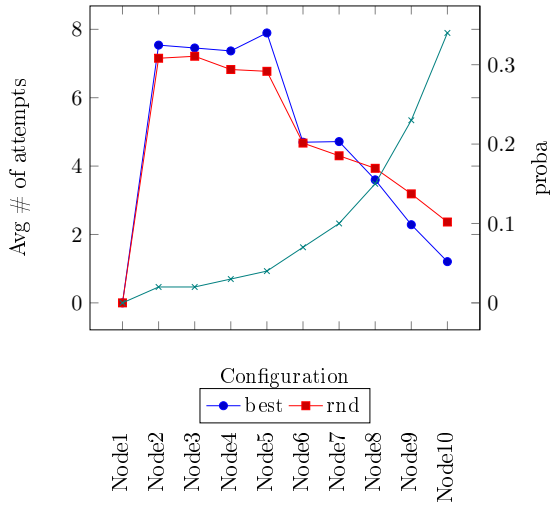


Figure V.14: Comparison of the avg # of attempts per Node in scenario *"k=1.5"*

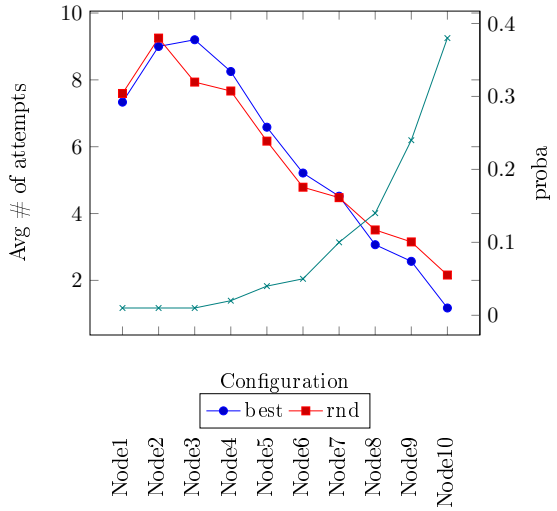


Figure V.15: Comparison of the avg # of attempts per Node in scenario *"k=1.6"*

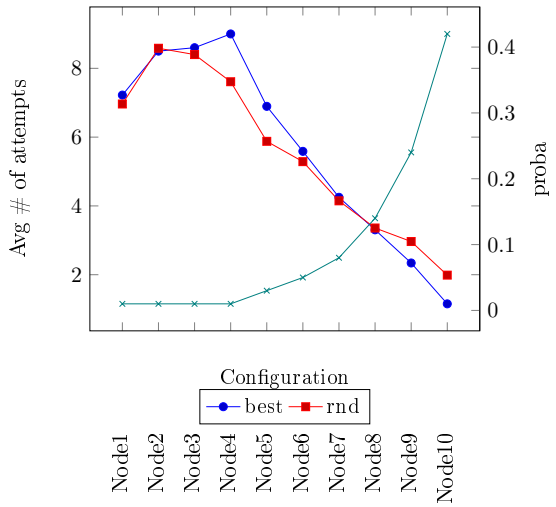


Figure V.16: Comparison of the avg # of attempts per Node in scenario *"k=1.7"*

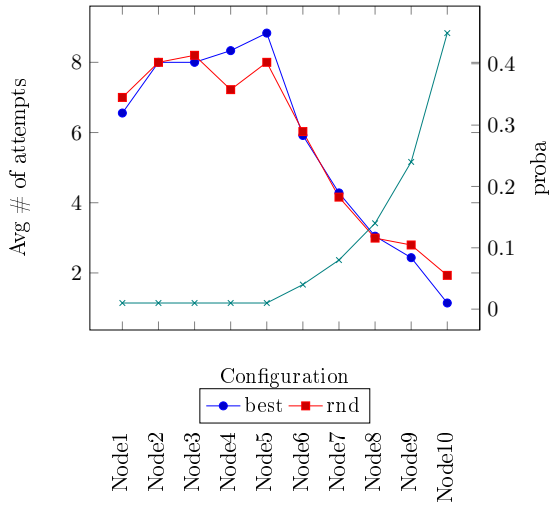


Figure V.17: Comparison of the avg # of attempts per Node in scenario *"k=1.8"*

k=1.9 Fig. V.18 shows the average amount of attempts per node when the probability difference between two consecutive Nodes increases by a factor 1.9

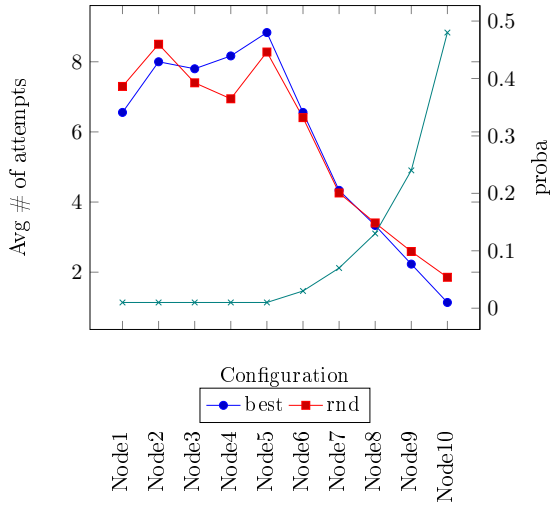


Figure V.18: Comparison of the avg # of attempts per Node in scenario *"k=1.9"*

One dominant Fig. V.8 shows the average amount of attempts per node when one of the Node has a much higher probability than the others.

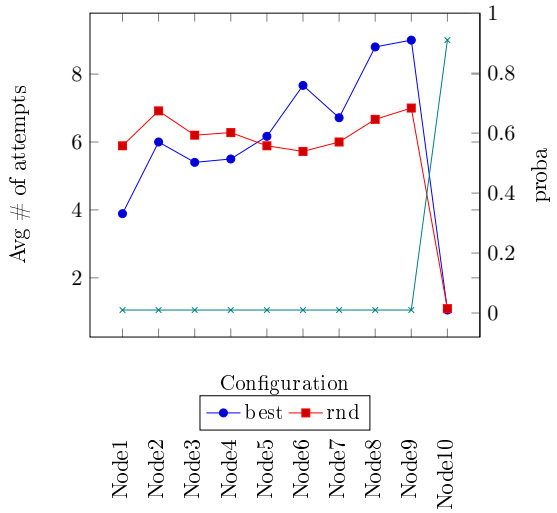


Figure V.19: Comparison of the avg # of attempts per Node in scenario *"1+"*

One inferior Fig. V.8 shows the average amount of attempts per node when one of the Node has a much lower probability than the others.

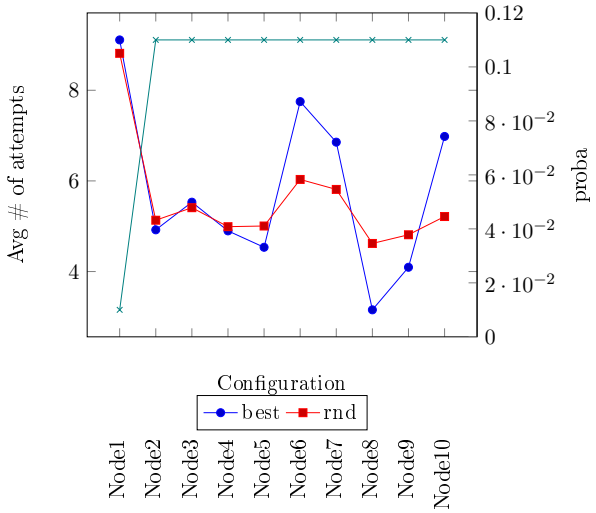


Figure V.20: Comparison of the avg # of attempts per Node in scenario *"1-"*

Quarter dominant Fig. V.8 shows the average amount of attempts per node when a quarter of the Node has a much higher probability than the others.

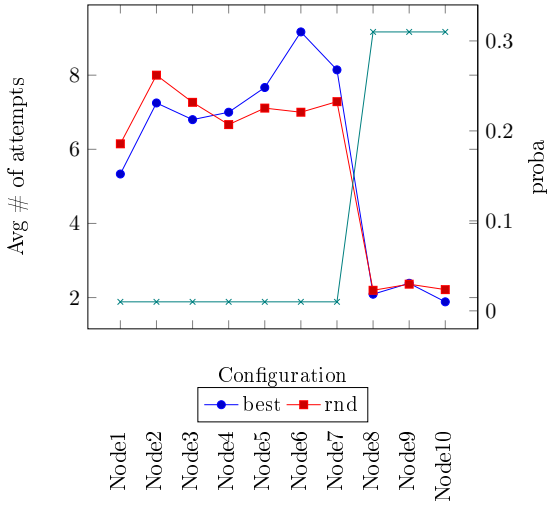


Figure V.21: Comparison of the avg # of attempts per Node in scenario *"1/4"*

The only conclusion that can be drawn out of those simulations is that it is irrelevant to choose one or the other algorithm, except if all the probabilities

Distribution	Average *"best"*	Standard deviation *"best"*	Average *"rnd"*	Standard deviation *"rnd"*
allEven	5.54	1.49	5.48	0.57
halfDominant	5.61	2.54	5.54	2.23
k.1.1	5.63	1.35	5.67	2.19
k.1.2	5.5	1.58	5.6	2.46
k.1.3	5.6	1.68	5.6	2.45
k.1.4	5.72	2.68	5.68	2.06
k.1.5	4.67	2.72	4.62	2.27
k.1.6	5.69	2.68	5.66	2.26
k.1.7	5.68	2.65	5.51	2.23
k.1.8	5.63	2.3	5.65	2.61
k.1.9	5.69	2.31	5.69	2.59
oneDominant	5.76	1.61	5.69	2.22
oneSub	5.78	1.74	5.58	1.15
quarterDominant	5.62	2.24	5.77	2.56

Table V.1: Average # of attempts and standard deviation in each scenario

are pretty even: in that case, one should prefer the algorithm using weighted probabilities.

V.2 Real case application

Phronesis is now being deployed on all the LHCb Online Cluster. One has to bear in mind that it is not a replacement to any solution already in place, but comes in addition to it. For example, the SCADA system which is used to control all the detector as well as the data flow comes with embedded recovery capacities: it would be a big mistake not to use them, as we have seen that a dedicated solution is always more efficient than a general approach. What the SCADA system does is to rely on a *"process manager"* which will make sure that all the SCADA processes are running properly, and attempts restart and likes if needed. Note that the process manager could be placed under the control of Phronesis.

LHCb, as well as all the other LHC related infrastructures, are at the time of this writing 2013 in an exceptional situation: all the equipment, as well as many software and methods, are being reviewed, reinforced, repaired or sometimes even replaced. This period, running from February 2013 till end of 2014 is called *"LS1"*, which stands for *"Long Shutdown 1"*. During those two years, the LHC and all the experiments are running in a stand-by mode and are upgraded in order to cope

with the energy increase of the LHC. In the environment of the LHCb Online, it consists of reshuffling the servers and network infrastructure, replacing a lot of hardware and redesigning the data flow, which translates into rewriting or adapting part of the software stack.
While thrilling because involving a lot of R&D in many areas, this stand-by mode has as a direct consequence for us, which is to lower the opportunities to test the software.

However, at the time of writing, a fair share of the LHCb Online system is already covered and the few diagnoses one had the opportunity to trigger showed useful.

V.2.1 Online system configuration

We shall now see the different systems that are known to Phronesis at the time of writing.

V.2.1.i Central Storage System

All the Online system makes a heavy use of a central storage system. The setup consists of a rack of disk trays with two active-active controllers that export all the devices to six servers. Those six servers are running a clustered file system, and are exporting the data via NFS for the linux clients ([Shepler 2003]), and Samba for the windows clients. The six servers, called *"store nodes"*, form together a cluster: if one of them is failing, another one takes the load and endorses the role of the dead node. In principle, all six nodes are equivalent. Some of the data that are shared like this are for example the user's home folders, or part of the software stack.

The cluster system and all the failover mechanism is managed by two packages called *"PaceMaker"* ([Beekhof 1999]) and *"Corosync"* ([CorosyncTeam 2008]). The role of PaceMaker is to manage the resources (such as an NFS server or a database instance) and make sure they are running in one and only one place (to avoid the split brain situation mentioned in section III.1). In order to achieve this, PaceMaker relies on Corosync, which is a messaging system: it makes sure that all nodes that belong to the cluster gets the communication and ensures synchronization.

Thus, thanks to those two software, the storage system already has recovery capacities. However, it sometimes happens that the migration of a service from one store node to the other is not successful, and it is in this situation that Phronesis can help.

Nevertheless, the management of the resources using PaceMaker is not standard, and makes use of special tools. Also, because PaceMaker and Corosync do not use

ASCII files to store their configuration and their status, it is extremely difficult to know what the present situation is without using the dedicated tools.

Given those constraints, the only help Phronesis can provide is to make sure that a resource (a process, most of the time) is running on one of the store node. If it is not the case, the *"command"* attribute of the ProcessAgent allows to give the command line that will instruct PaceMaker to start the resource, instead of suggesting the default solution. This diagnosis and recovery suggestion are correct.

Another type of resources are IP addresses: some IP addresses are moved from one node to the other. The aim of it is to keep the client agnostic to the cluster configuration. For this type of resource, Phronesis unfortunately does not behave properly. In fact, the only thing one can do is to define those floating addresses as Servers. If one of them is not assigned to any of the store node, Phronesis will detect the associated Server as down, and suggest to start it up. This diagnosis is not really clear, and not even exact: it is a flaw of Phronesis. On the other hand, one could argue that floating IP addresses are more of a network concern, which were consciously discarded from the beginning.

Finally, Phronesis can make sure that all store nodes are running the required processes to have Corosync and PaceMaker fully functional.

V.2.1.ii CVMFS

CVMFS ([CVMFSTeam 2010]) is the *"cernVM file system"*. It is a way to distribute file over a large number of client. The main idea is that when a client tries to access a file, it will receive it from a central server, and keep it in a local cache. This means that the clients have no write access on the files. Note that it is very different from the behavior of an NFS share: while one gets a local copy of a file with CVMFS, with NFS one directly instructs the server to perform the action on the file.

CVMFS is installed on the Online cluster for less than a month at the time of writing, and is meant to replace NFS in exporting the software stack to the nodes.

Phronesis is configured to diagnose the client side (all needed processes running, correct configuration, cache directory correct), as well as the server side. In fact, the LHCb Online does not have a real CVMFS server: the files are hosted somewhere else at CERN, where the LHCb Online team has no control. However, the store nodes are running proxies that forward the client requests to the real server, and Phronesis can diagnose this (with the same limitations as for the central storage, since it is also managed with PaceMaker).

The case of CVMFS is interesting in terms of dependency between the clients

and the servers: indeed, the client will totally depend on the server the first time it will access a file; however, because of the cache mechanism, even if the server then fails, the client will still be able to access the file. So in fact, it is a *"temporary dependency on a per file basis"*. To avoid any confusion, the client is considered dependent on the server.

V.2.1.iii CASTOR

CASTOR ([CastorTeam 1999]) is the *"Cern Advanced STORage manager"*. It provides a way to manipulate to a certain extent the files that are on tape. Because the commands to manipulate CASTOR are different from the standard Linux one, castorfs was developed. Castorfs is a Fuse file system which allows to browse the files on CASTOR just like one would browse a folder on a standard file system (like ext3, btrfs or else).

Phronesis is configured to make sure that the client side is working properly. However, as for CVMFS, the LHCb Online team has absolutely no control on the server side, which is totally unknown to Phronesis.

V.2.1.iv DIM

DIM ([Gaspar 1993]) is the *"Distributed Information Management system"*, another piece of software developed at CERN. DIM is a communication system for distributed / mixed environments, it provides a network transparent inter-process communication layer. It is very heavily used in LHCb for various purposes, but primarily for control and monitoring.

The idea of DIM is to give to processes an easy way to offer a so-called *"Service"* on the network (e.g.: publishing the value of a counter when it changes, or at regular interval), and an easy way to *"subscribe"* to them (*e.g.*: triggering a callback function when the value of the counter has changed). The processes that want to subscribe to a service do not know at first where the service runs, and they do not not need to: central to this system is the *"DIM DNS"*, which is like a catalog where all the processes can declare themselves as offering a service, and where all the processes can see which services are offered. So when a process wants to subscribe to a given service, it will make the request to the DIM DNS, which will forward it to the process offering the service. Once the connection is established, the DIM DNS is not involved anymore.

Phronesis is configured to make sure that the DIM DNS is running properly, and that the processes have the address of the correct address for the DIM DNS: there is a full net of them in the LHCb Online environment, and a process that will

contact the wrong one will not find the service it is looking for.

V.2.1.v Event filter

The primary goal of the Online farm is to run the High Level Trigger, that is a selection process ([Alessio 2008]). The software stack developed for it is sometimes referred as *"event filter"*. It consists of a handful of processes running on each farm node, that Phronesis is configured to diagnose.

For historical reasons, all those processes are not started in a standard Linux way, but are managed by a custom made process manager. This process manager normally takes care of restarting dead processes. Once again, the *"command"* attribute of the ProcessAgent shows useful in this situation.

V.2.1.vi Web Services

All the web services are currently under the control of Phronesis. Until very recently, the web service infrastructure was exactly as described in the simulation V.1.1, in which Phronesis has proved to be efficient. In fact, the first version of Phronesis was tested against the real web service infrastructure before the conference CHEP ([Haen Christophe 2012]), and the diagnostics were always correct, which confirms the results of the simulation.

However, the NFS share for the web servers was recently changed: it is not anymore relying on a standard server to export the files via NFS, but on a proprietary hardware which behaves as a black box for the user. This means that Phronesis cannot be used to diagnose the NFS server. Nevertheless, this proprietary solution is known to be very reliable, and one can expect the NFS service provided by it to be very rarely the root cause of problems.

V.2.1.vii Databases

The LHCb Online environment runs a certain amount of databases. Some of them are Oracle, on which the LHCb Online team does not have any control. But the environment runs several MySQL servers ([Corporation 2013]), as well as one MariaDb server ([Foundation 2013]), which is a fork of MySql . Those database instances are configured in Phronesis: it contains information about the processes and the files needed by the database instance, just like in the simulation.

The case of MySql is one of those where Phronesis really shows useful: the processes are started in a standard way using linux services, the configuration is stored in standard locations in ASCII files, there is no embedded feature of

diagnosis or advanced recovery, *etc.* However, note that Phronesis cannot detect failure that are inherent to the application itself: if for example a table is corrupted, all the services using this table will be impacted, but Phronesis will not be able to notice it out of the box.

A possibility for this particular case would be to define a FileAgent which parses the log file for lines containing *"corrupted"*. In this case, Phronesis would report that the log file contains an unwanted pattern, which is a correct diagnosis. But the recovery solution it would offer would be to modify the file content, which is of very little help here. However, since it is possible for a user to leave comments, one can always associate a comment which explains the real recovery procedure. Finally, note that one would need to rotate the log file, or remove the *"corrupted"* line from the log file in order to avoid fake diagnoses because of ancient problems.

V.2.1.viii Icinga

The monitoring system infrastructure is based on Icinga, as mentioned in chapter II. However, Icinga used out of the box is not able to cope with large environments like the LHCb one. One thus needs to put in place a much more complex monitoring system, as described in [Haen C 2011].
To summarize the setup:

- a central server running Icinga, Ido2db to write the monitoring data into a database (a local MySql instance), gearmand (which manages in-memory queues in which Icinga puts checks to execute).
- many servers running a gearman worker that fetches tasks to execute from the gearmand server, and puts the result back into a special queue.

An NFS server running on the central server is also used to share files with all the workers, but this is meant to be replaced very soon.

Phronesis is configured to diagnose all of this monitoring setup: it involves many configuration files and many processes running on a large number of servers, with a lot of dependencies between each other.

V.2.1.ix Log servers

Keeping track of what is happening or what has happened is very important to troubleshoot problems. Since it can be cumbersome, and sometimes not possible, to go on each and every machine to read the logs, LHCb has put in place a central logging system [J.C. Garnier 2011]. It consists of several machines running in a cluster mode similar to the one of the store nodes, and running many different processes that are meant to collect and gather the logs from all over the Online

system.

Phronesis has been configured to diagnose problems on those various processes, as well as on their configuration files.

V.2.1.x Still missing

At the time of writing, there is still a certain amount of systems that could be placed under the control of Phronesis but that have not been done yet.

It is for example the case of the virtualization infrastructure, which is more and more developed [E. Bonaccorsi 2011]. In fact, all the servers running web services are now virtualized, so it would be good to have the virtualization infrastructure as a dependency of the web services. Even though the question was not analyzed in depth, tracking down a problem to a failure in the virtualization system promises to be very challenging because it works with similar concept of failover and redundancy as PaceMaker.

Another example would be a new performance monitoring system recently put in place called *"Ganglia"* [Project 2013a]. The role of Ganglia is to propagate in a tree like structure information about performances in order to plot graphs on a central head node. Finding why the plots for a given node are empty is typically the use case in which Phronesis can show useful.

V.2.2 Feedback from the real case application

As mentioned at the beginning of this chapter, LHCb is in a standby mode with many downtimes, and despite the fact that there were not so many opportunities to test Phronesis on real unexpected and not provoked situations, we could still learn quite a few interesting lessons.
First of all, on the positive side, it is to be noted that Phronesis could place several diagnoses that turned out to be correct, and offer appropriate solutions. Among these, several diagnoses were a direct consequence of the Convention Over Configuration approach taken, because the root cause was pointing at elements the user did not defined manually.

Examples of diagnoses are:

- Full inodes for the log servers: the log servers store on a clustered file system a huge amount of tiny files. As a consequence, the pool of inodes ran out way before the actual storage space did. The solution, correctly suggested by Phronesis, was to remove files. In fact, this problem was spotted before it actually happened because of the default threshold set to 99% of used inodes:

it is a great chance, because otherwise all the new logs that would have required a new file would have been silently lost.

- Bad mount options on a web service: one of the web service required a particular folder to be mounted with the write option, which was not the case. Phronesis suggested to remount it with the appropriate option. Although correct, this would not have worked immediately, because the *"black box NFS server"* on which Phronesis has no control was not configured to accept it.

- Incorrect DIM DNS address: the file containing the information was corrupted

- Various problems on MySql servers: running out of disk space and error in the configuration files were among the problems diagnosed by Phronesis on the MySql database

- Various problem on the monitoring infrastructure: the mail alerts not being sent tracked down to a process not running, the results out of date tracked down to a full disk space and checks not executed because of some gearman workers not running are a few issues that Phronesis correctly diagnosed.

Of course, in some cases, Phronesis completely missed the root cause of the problems. This was either because of situation not forecasted in the design (see section VI.1) or because of an incomplete configuration. When it was the latest, the configuration was always updated to make sure to cover that case in the future. When it was due to a non expected situation, the code was improved if it did not imply too heavy modifications in order to avoid destroying what already works; other cases were left on the *"todo list"* for future developments.

One of the really positive surprises was how useful the principle of Convention Over Configuration proved to be. In general, although more verbose that what we would have wished at the beginning, the configuration grammar described in section IV.2.1.ii is quite satisfying. Some flaws, addressed in chapter VI , were found when dealing with this large configuration, and they were corrected when possible. The Shared Experience principle was very heavily used: one defined from the beginning very abstract templates for web services, for NFS, for databases and more, that were specialized more and more to the LHCb environment using inheritance, up to the point where configuring a new instance of them is about five lines. There were not enough cases to diagnose to benefit from the Shared Experience at the diagnosis level.

In terms of raw performances (CPU and memory consumption, running time, *etc*), improvements can be addressed in many places (see chapter VI).

The parser needs about 7mn, 100% CPU on one core, and 3 Gb of memory to treat the almost 9 000 lines of configuration for the online environment and produce

almost 30 000 MetaAgents.

The footprint of the Core on the machine is pretty low, even during a diagnosis, however it needs about 30 minutes to check every single of those 30 000 MetaAgents. This long time is explained by the fact that they are treated sequentially, and also because each MetaAgent timing out will take 20 seconds. Also, recall that Phronesis is not meant to check so many MetaAgents, but rather to diminish the amount it needs to check before finding the culprit.

The footprint of the RemoteAgent on a server is completely negligible.

Chapter VI

Future

Contents

Despite our software already proved useful in certain situations, we already see many improvements that would be possible for it, both in terms of functionalities and technicalities.

VI.1 Technical aspects

The development of Phronesis and all its software stack was done by a single person. Also the final results counts only about 25 000 lines of code — taking into account only what is needed to run the Core and the Remote Agent, and not the simulation tool or the various API implementations — it has taken many more to reach this point. Because of mainly time constraints, some aspects of the development were taken a bit lightly.

This is especially the case of all the testing functionalities: there is no unit tests, nor regression tests, nor integration tests. Adding these, or at least cover the most central parts of the code, would be well invested time for future developments.

The error handling in Phronesis is based on the exception mechanism of C++. Although it was tried during the development to cover as many situations as possible, there are still some places where exceptions are not handled, and thus leading to an abortion of the program. Those bugs are corrected as soon as they are detected. Another type of error handling which is missing in Phronesis is the advanced control of the data: for example, there is hardly any protection against buffer overflow.
Nevertheless, great care has been taken and relatively complex implementations were done in the Core and the Remote Agent in order to avoid any kind of deadlock that could result from network communication glitches or system commands hanging.

The configuration parser is the place where big improvements are the easiest to do. First of all, the running time and the memory consumption could be greatly reduced. A profiling of the code running against the configuration of the Online environment showed that about 50% of the running time was spent making copies of objects. Those copies are the result of the inheritance mechanism. A behavior that would greatly improve the situation is a *"copy-on-write"* mechanism. The principle is, when performing a copy, to actually points to the original memory location, and only performing the real copy and duplicating the memory when a write operation is to be executed. This is for example what happens behind the scene in C++ with string objects. Unfortunately, Python does not have such a behavior and one needs to perform it manually.
In fact, we already have a technical solution to implements this behavior, without modifying too heavily the existing code, but it is not implemented yet because of time constraints. The idea is to use a pool of values, a bit like the flyweight design pattern does, and to give to the objects the position of their values in the pool instead of the actual value.

Let's illustrate the principle: suppose, as visible in Fig. VI.1, a class *"A"*, which has two attributes, for example integers, *"u"* and *"v"*. If one has an instance X(u = 1, v = 2), on which one performs a deep copy to get the instance Y(u = 1, v = 2), and then updates *"v"* to get Y(u = 1, v = 4), four memory locations will actually be used to store the values of both *"u"* and both *"v"*.
If instead of this, the possible values for *"u"* and *"v"* are kept in a pool of values, the X and Y attributes just have to point to those values. If all the attributes were integers, it would not be interesting, because indexes are integers. However, for any bigger type, string values for example, it shows extremely efficient.
The great advantage of this solution in our case is that there is no need at all to change the logic of the algorithms. One needs to replace attributes of the classes with index pointing to the pool of values, and replace the setter and getter methods to actually act on the pool rather than the object itself; but all the rest, including the calls to deep copy, will remain exactly the same.

Another technical improvement that should be done on the parser is the error handling. While program aborts when it should, the exception handling is not done properly enough to be able to provide the user with detailed explanations on the cause. For example, if a Coordinator X requires a Coordinator Y that does not exist, a message *"Coordinator Y does not exist"* will be printed, while *"Coordinator Y required by Coordinator X at line n does not exist"* would be more useful.

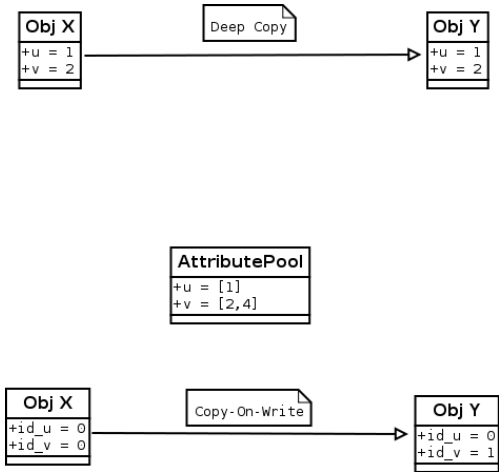


Figure VI.1: Illustration of the Copy-On-Write mechanism

VI.2 Functional aspects

There are certain functionalities that we have been thinking of, and that were not implemented, once again because of time constraint.

To start with the parser, some extension to the grammar would be very useful, but challenging technically. Here are some examples:

- Reverse Trigger: for the time being, the Triggers are defined only in one direction. For example, if a modification of a file requires a process restart, one has to define the Trigger on the FileAgent. One would like the possibility to define it on the ProcessAgent, or eventually in the Coordinator that contains both of them.
- Dependencies on attached Coordinators are not supported yet. Although not crucial, this feature would allow a faster diagnosis, since the attached Coordinators are checked at the end only
- If a line is not understood by the grammar, it is silently ignored, which is not good. The parsing should stop and print error messages.
- A useful feature would be to support variable in part of the attribute. For example *"content -> my file content with a $myVar that I define later;"*, or *"filename -> /my/base/path/$varFile;"*
- For the time being, variables cannot be reassigned. Once a variable was assigned, it is forgotten by the MetaAgent, and thus cannot be overwritten.
- Apply group treatment: one of the reason which makes the grammar more verbose than one would like is because there is no *"group"* notion. If for

example one wants to define a Coordinator for each servers in which one just inherits an abstract template and sets the server attribute, one needs to write three lines as many times as servers. It would be useful to define groups and macros to apply the loop by itself.

The improvements at the Core level consist of adding checks and tests to cover more situations. Some examples are:

- Check disk health
- Add test on the uid and gid of files and processes
- Check open ports: this, as well as other checks like iptable rules, are half way between system and network, and were left aside so far.
- Add constraint that are evaluated dynamically: the grammar should for example support statement like *"user readable"* or *"group writable"*, and the Core should be able to cope with it
- Deal better with clusters: although there is no concrete plan for it, clusters should be better taken into account.
- Custom Agents: as we have seen all along this document, Phronesis works primarily with files and processes, which is sufficient for most cases. However, situations which are hardly manageable with those entities were mentionned, such as the Corosync and PaceMaker tools. To treat such cases, one could give the possibility to users to develop their own pieces of code to define custom agent. The technical solution for this would be to use dynamic libraries loaded at run time for the Core, use meta-tables in the database (tables that describe tables), and use the flexibility of python for the compiler. Even though way too complicated to be fully implemented in a reasonable time, a proof of concept for the Core could be developed and tested. No attempt were made to make the compiler support dynamic grammar.

While there was no need yet for it, given the growing popularity of Python, an attempt to write a wrapper of the API in Python was made. Unfortunately, the multi threaded implementation of the API makes a simple wrapper impossible, and one would need to develop a native Python API. No time was invested in this at the time of writing.

More than real improvements to the existing software, we felt the need of integration with other tools to lower even more the work of the system administrator in configuring Phronesis. The idea was to develop a tool which will, given Phronesis configuration files containing abstract entities as well as some directives, extract information from the cluster management software and produce Phronesis configuration files containing instantiations of those abstract entities. This project was given to a student that worked on it for about six months. Unfortunately, it

never came to work, and one would need to start it all over again.

A possibility that was considered to be feasible but has not been implemented was to add very simple algorithms to forecast problems based on historical data. Since Phronesis is idle most of the time, a possibility is to collect data — such as disk space, swap consumption, and so on — on a regular basis on all the servers, and using simple interpolation mechanisms, forecast soon to come problems. One could also notice if a value has changed compared to usually: the owner of a file different from what it used to be, or a process CPU consumption way off the usual value, *etc.* From a pure technical point of view, this could be easily implemented in a separate thread, without perturbing the normal behavior of Phronesis.

One feature which is currently missing in Phronesis and will probably show useful — or even needed — in the future is the possibility to receive inputs from multiple sources. This means that one monitoring system would report a given set of problems, while another system would report other problems. This situation is very realistic, since it is also what happens to a human expert: various users are reporting problems depending on the systems they are using, and if a failure impacts several users, they will all report it to you more or less simultaneously. While from a pure technical perspective, Phronesis is able to receive informations from multiple sources, its logic is absolutely not adapted to such a case. This comes from the dependency inference mechanism: it is the difference between two consecutive reports that gives new rules. So if the reports come from two different systems that are monitoring different services, all the logic will fail and induce errors in the rules. One way to deal with this would be to track the sources of the problem report: if they are different, reported problems are added to the existing problems. This represents a great technical challenge — that would require, among other things, a different approach for the threads and locking management — but also deep changes in the logic of Phronesis. It would really be a major change.

Finally, there is currently no improvement of the recovery procedure offered by Phronesis. While its diagnosis improve with time, the recovery solution remain the same, and are just enhanced with historical data and user comments on previous cases. It might be interesting to see whether some kind of reinforcement learning phase could be applied here as well. Another potential approach would be to apply association rule learning algorithms mentioned in section IV.3.1.iv on the history data that are kept.

Chapter VII

Conclusion

This work aims at supporting system administrators in their task of managing large computer environments, requiring a very broad range of competences. We have seen that users are directly exposed to the availability of a data center, and that a high availability can be reached only at a very high cost. Not all the data centers can afford a full-time on-site expert team, and this leads to have on-call experts. However, this incurs a certain reaction time, and the diagnosis and recovery process of a problem are not always realized in optimal conditions, which results in a lower availability. We have enumerated several properties a tool should have in order to assist the experts:

- reduce the workload of the experts by requiring as little maintenance and configuration as possible
- offer a diagnosis to problems, as well as a recovery solution
- improve with experience
- act as knowledge base to centralize as much information as possible about the computer environment. This is mandatory in order to propose diagnoses.
- act as a problem history base in order to keep encountered situations at the disposal of experts

Chapter 2 documents the various attempts that were made in order to address such problems:

- classical monitoring: this involves having as many probes as possible in the environment. We have seen that it is not self-sufficient, but still an essential element in any infrastructure.
- expert system: consisting of a set of rules and an inference engine that will follow the rules in order to draw conclusion. It has proven to be extremely successful in many areas, but not adapted to broad domains such as system administration.
- autonomic computing: introduced by IBM and implemented using the MAPE-K loop, autonomic computing requires giving self management capacities to each element of an environment. We have seen that it shows high efficiency when the autonomic behavior is implemented directly in each element, and taken as a constraint from the beginning. Adding autonomic behavior a posteriori is very challenging.

This work tried to mimic the exact reflection process of a human expert in order to provide a software as generic as possible to match the heterogeneous environment of the LHCb experiment. To do so, we have adapted various methodologies from the literature, and offered original approaches:

- be totally non-intrusive for the existing components of the environment by focusing on Linux systems and using files and processes as the building blocks for our diagnoses.
- perform no active monitoring but wait to be informed of problems.
- adapt and use a single MAPE-K loop to control all the elements instead of one per element. This allows the software to infer dependency rules between elements.
- use reinforcement learning to improve the diagnosis speed and increase the scalability.
- use the Shared Experience principle to overcome the limitations of reinforcement learning and reduce the workload of administrators, together with the Convention Over Configuration principle.
- offer a default recovery action for each standard problem, and build a full recovery procedure out of it.

Putting into practice all these methods required the development of various pieces of software, as well as the definition of a new grammar for the system administrator to express the environment setup. We wrote a compiler to read, interpret and load the configuration files into a database without loosing the experience previously gained. A small agent running on every node to make all the measures required for diagnosis was implemented. Finally, the Core software contains all the algorithms that mimic the behavior of a human expert to provide the user with diagnosis and recovery procedure. An API was also developed to export the functionalities to third party code. The result is called Phronesis.

Phronesis was tested in a first period against simulations:

- the first simulation showed the ability of our software to address situations similar to those encountered in many environments, including LHCb. It also illustrates the importance of the dependency rules, as well as the capacity of Phronesis to discover them.
- the second simulation proved that the Shared Experience principle has a big impact in terms of performance and scalability, and that it improves the learning speed of the reinforcement learning algorithms.
- the last simulation was meant to compare the two different exploration strategies discussed in Chapter 4. The conclusion is that there is hardly any difference.

The simulations confirmed that our methods were successful in various cases, and we could then apply Phronesis to the LHCb environment. At the time of writing, all the LHC experiments are in a consolidation phase, which means that Phronesis could not be tested in real running situation. However, there were still several opportunities for our software to prove useful: it made several correct diagnoses and offered appropriate recovery solutions. A very positive surprise from the real case application was the efficiency of the Convention Over Configuration principle: several diagnoses Phronesis made were a direct consequence of the principle. The interest of the Shared Experience principle in terms of performance was not visible because of the low number of problems Phronesis diagnosed, but it proved very useful when writing the description of the environment.

Together, the simulations and the application on the LHCb environment proved that Phronesis can be helpful for system administrators and fulfill the goals we originally set. They also illustrate that the various aspects of our approach — like the concept of reinforcement learning, the unique MAPE-K loop or Shared Experience — are efficient, and that they can, when used together, mimic to a certain extend the diagnosis process of a human expert.

However, there is still large room for improvement, both in terms of the technical implementation and of functionality. This includes, for example, an extension of the grammar, which is unfortunately more verbose than what we hoped at the beginning; better native support for cluster systems; and dynamic constraints on the properties of files and processes.

Phronesis was mentioned at OSMC 2012, the Open Source Monitoring Conference. It received great attention, and developers of open source projects showed an interest in participating in the development of Phronesis. We hope to be able to release it as an open source solution that the community would pick up, and extend it following the recommendations made in chapter VI.

Bibliography

[Agrawal 1993] Rakesh Agrawal, Tomasz Imielinski and Arun Swami. *Mining association rules between sets of items in large databases.* SIGMOD Rec., vol. 22, no. 2, pages 207–216, June 1993. (Cited on page 70.)

[Agrawal 1995] Rakesh Agrawal and Ramakrishnan Srikant. *Mining Sequential Patterns.* In Proceedings of the Eleventh International Conference on Data Engineering, ICDE '95, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society. (Cited on page 70.)

[Alessio 2008] F Alessio, C Barandela, L Brarda, M Frank, B Franek, D Galli, C Gaspar, E v Herwijnen, R Jacobsson, B Jost, S Kostner, G Moine, N Neufeld, P Somogyi, R Stoica and S Suman. *LHCb Online event processing and filtering.* Journal of Physics: Conference Series, vol. 119, no. 2, page 022003, 2008. (Cited on page 97.)

[Ananthanarayanan 2005] Rema Ananthanarayanan, Mukesh K. Mohania and Ajay Gupta. *Management of Conflicting Obligations in Self-Protecting Policy-Based Systems.* In ICAC, pages 274–285, 2005. (Cited on page 26.)

[Barrett 2004] R. Barrett, P.P. Maglio, E. Kandogan and J. Bailey. *Usable autonomic computing systems: the administrator's perspective.* In Autonomic Computing, 2004. Proceedings. International Conference on, pages 18 – 25, may 2004. (Cited on page 19.)

[Beekhof 1999] Andrew Beekhof. *PaceMaker website*, 1999. (Cited on page 94.)

[Boost-team 2013] Boost-team. *Boost libraries*, 2013. (Cited on page 54.)

[Brambilla 2012] Marco Brambilla. Model-driven software engineering in practice. Morgan & Claypool Publishers, 2012. (Cited on page 19.)

[Buşoniu 2006] L. Buşoniu, B. De Schutter and R. Babuška. *Decentralized reinforcement learning control of a robotic manipulator.* In Proceedings of the 9th International Conference on Control, Automation, Robotics and Vision (ICARCV 2006), pages 1347–1352, Singapore, December 2006. (Cited on page 27.)

[CastorTeam 1999] CastorTeam. *CASTOR website*, 1999. (Cited on page 96.)

[Cesa-Bianchi 1997] Cesa-Bianchi, Freund Yoav, Haussler David, Helmbold David P., Schapire Robert E. and Warmuth Manfred K. *How to use expert advice.* J. ACM, vol. 44, no. 3, pages 427–485, May 1997. (Cited on page 31.)

[Chan 2003] Hoi Chan and Trieu C. Chieu. *An approach to monitor application states for self-managing (autonomic) systems.* In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '03, pages 312–313, New York, NY, USA, 2003. ACM. (Cited on page 19.)

[Charles X. Ling 2004] Jianning Wang Charles X. Ling Qiang Yang and Shichao Zhang. *Decision trees with minimal costs.* In Proceedings of the twenty-first international conference on Machine learning, ICML '04, pages 69–, New York, NY, USA, 2004. ACM. (Cited on page 31.)

[Chen 2009] Yen-Liang Chen, Mi-Hao Kuo, Shin-Yi Wu and Kwei Tang. *Discovering recency, frequency, and monetary (RFM) sequential patterns from customers purchasing data.* Electron. Commer. Rec. Appl., vol. 8, no. 5, pages 241–251, October 2009. (Cited on page 70.)

[Cheng 2006] Shang-Wen Cheng, David Garlan and Bradley Schmerl. *Architecture-based self-adaptation in the presence of multiple objectives.* In Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, SEAMS '06, pages 2–8, New York, NY, USA, 2006. ACM. (Cited on pages 18 and 19.)

[Chomsky 1956] Noam Chomsky. *Three models for the description of language.* IRE Transactions on Information Theory, vol. 2, pages 113–124, 1956. `http://www.chomsky.info/articles/195609--.pdf` – last visited 14^{th} January 2009. (Cited on page 127.)

[Chomsky 1957] Noam Chomsky. Syntactic structures. Mouton & Co., 1957. (Cited on page 127.)

[Cobleigh 2002] Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise and Barbara Staudt Lerner. *Containment units: a hierarchically composable architecture for adaptive systems.* In Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '02/FSE-10, pages 159–165, New York, NY, USA, 2002. ACM. (Cited on page 25.)

[CorosyncTeam 2008] CorosyncTeam. *Corosync website*, 2008. (Cited on page 94.)

[Corporation 2013] Oracle Corporation. *MySQL website*, 2013. (Cited on page 97.)

[Craven 2008] B. Settles M. Craven and L. Friedland. *Active learning with real annotation costs.* In NIPS Workshop on Cost-Sensitive Learning, page 1–10, 2008. (Cited on page 31.)

[CVMFSTeam 2010] CVMFSTeam. *CVMFS website*, 2010. (Cited on page 95.)

[D. A. Cohn 1996] Z. Ghahramani D. A. Cohn and M. I. Jordan. *Active Learning with Statistical Models.* In Journal of Artificial Intelligence Research, Volume 4, pages 129–145, 1996. (Cited on page 31.)

[Diao 2005] Y. Diao, J.L. Hellerstein, Sujay Parekh, R. Griffith, G. Kaiser and D. Phung. *Self-managing systems: a control theory foundation*. In Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the, pages 441 – 448, april 2005. (Cited on page 19.)

[Dolev 2008] S. Dolev and R. Yagel. *Towards Self-Stabilizing Operating Systems*. Software Engineering, IEEE Transactions on, vol. 34, no. 4, pages 564 –576, july-aug. 2008. (Cited on page 19.)

[Duda 2000] Duda and Hart. Pattern classification. Wiley-Interscience, 2000. (Cited on page 65.)

[E. Bonaccorsi 2011] M. Chebbi E. Bonaccorsi L. Brarda and N. Neufeld. *Virtualixation for the LHCb experiment*. In Proceedings of ICALEPCS2011, pages 1157–1160, 2011. (Cited on page 99.)

[El-Sayed 2004] Maged El-Sayed, Carolina Ruiz and Elke A. Rundensteiner. *FS-Miner: efficient and incremental mining of frequent sequence patterns in web logs*. In Proceedings of the 6th annual ACM international workshop on Web information and data management, WIDM '04, pages 128–135, New York, NY, USA, 2004. ACM. (Cited on page 70.)

[Enterprises 2009] Nagios Enterprises. *Nagios website*, 2009. (Cited on pages 15 and 16.)

[Foundation 2013] MariaDB Foundation. *MariaDb website*, 2013. (Cited on page 97.)

[Garlan 2004] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl and P. Steenkiste. *Rainbow: architecture-based self-adaptation with reusable infrastructure*. Computer, vol. 37, no. 10, pages 46 – 54, oct. 2004. (Cited on page 18.)

[Gaspar 1993] Clara Gaspar. *DIM website*, 1993. (Cited on page 96.)

[Ginsberg 1993] Matt Ginsberg. Essentials of artificial intelligence. Morgan Kaufmann, 1993. (Cited on page 16.)

[Greiner 2002] Russell Greiner, Adam J. Grove and Dan Roth. *Learning cost-sensitive active classifiers*. Artificial Intelligence, vol. 139, no. 2, pages 137 – 174, 2002. (Cited on page 31.)

[Haen C 2011] N. Neufeld Haen C E. Bonaccorsi. *Distributed monitoring system based on Icinga*. In Proceedings of ICALEPCS2011, pages 1149–1152, 2011. (Cited on pages 16 and 98.)

[Haen Christophe 2012] Bonaccorsi Enrico Haen Christophe Barra Vincent and Neufeld Niko. *Artificial Intelligence in the service of system administrators*. May 2012. (Cited on page 97.)

[Harrington 2002] D. Harrington. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks.* Rapport technique, Network Working Group, 2002. (Cited on page 15.)

[Hipp 1998] Jochen Hipp, Andreas Myka, Rüdiger Wirth and Ulrich Guntzer. *A New Algorithm for Faster Mining of Generalized Association Rules.* In In Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD '98, pages 74–82. Springer, 1998. (Cited on page 70.)

[Hofmeister 1992] Christine Hofmeister and James Purtilo. *Dynamic reconfiguration in distributed systems: adapting software modules for replacement.* Rapport technique, College Park, MD, USA, 1992. (Cited on page 25.)

[Horn 2001] Paul Horn. *IBM's Perspective on the State of Information Technology*, 2001. (Cited on page 17.)

[Huebscher 2008] Markus C. Huebscher and Julie A. McCann. *A survey of autonomic computing degrees, models, and applications.* ACM Comput. Surv., vol. 40, pages 7:1–7:28, August 2008. (Cited on page 17.)

[IBM 2001] IBM. *An architectural blueprint for autonomic computing*, May 2001. (Cited on page 17.)

[IEEE 2013] IEEE. *IEEE Std 1003.1, 2013 Edition, POSIX definition*, 2013. (Cited on page 33.)

[J.C. Garnier 2011] N. Neufeld J.C. Garnier L.Brarda and F. Nikolaidis. *LHCb online log analysis and maintenance system.* In Proceedings of ICALEPCS2011, pages 1228–1231, 2011. (Cited on page 98.)

[Jennings 2000] Nicholas R. Jennings and Michael Wooldridge. *On agent-based software engineering.* Artificial Intelligence, vol. 117, pages 277–296, 2000. (Cited on page 26.)

[Ji 2007] Shihao Ji and Lawrence Carin. *Cost-sensitive feature acquisition and classification.* Pattern Recogn., vol. 40, no. 5, pages 1474–1485, May 2007. (Cited on page 31.)

[Kaiser 2003] G. Kaiser, J. Parekh, P. Gross and G. Valetto. *Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems.* In Autonomic Computing Workshop. 2003. Proceedings of the, pages 22 – 30, june 2003. (Cited on pages 18 and 25.)

[Kamoda 2005] Hiroaki Kamoda and Krysia Broda. *Policy conflict analysis using free variable tableaux for access control in web services environments.* In In Policy Management for the Web Workshop, pages 5–12, 2005. (Cited on page 26.)

[Kanal 2000] Laveen N. Kanal. *Perceptron.* In Encyclopedia of Computer Science, pages 1383–1385. John Wiley and Sons Ltd., Chichester, UK, 2000. (Cited on page 32.)

[Karsai 2001] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon and Tamas Kovacshazy. *An Approach to Self-Adaptive Software based on Supervisory Control.* In In 2 nd International Workshop in Self-adaptive software, (IWSAS-01), Robert Laddaga, Howard Shrobe, and Paul, 2001. (Cited on page 19.)

[Kephart 2003] J.O. Kephart and D.M. Chess. *The vision of autonomic computing.* Computer, vol. 36, no. 1, pages 41 – 50, jan 2003. (Cited on page 17.)

[Khan 2008] Fahad Shahbaz Khan, Saad Razzaq, Kashif Irfan, Fahad Maqbool, Ahmad Farid, Inam Illahi and Tauqeer Ul Amin. *Dr. Wheat: A Web-based Expert System for Diagnosis of Diseases and Pests in Pakistani Wheat.* In Proceedings of the World Congress on Engineering 2008, 2008. (Cited on page 17.)

[Khan 2011] Abdur Rashid Khan, Hafeez Ullah Amin and Zia Ur Rehman. *Application of Expert System with Fuzzy Logic in Teachers' Performance Evaluation.* February 2011. (Cited on page 17.)

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier and John Irwin. *Aspect-oriented programming.* In ECOOP. SpringerVerlag, 1997. (Cited on page 19.)

[Kon 1999] Fabio Kon, Roy H. Campbell, M. D Mickunas and Klara Nahrstedt. *2K: A Distributed Operating System for Dynamic Heterogeneous Environments.* Rapport technique, Champaign, IL, USA, 1999. (Cited on page 25.)

[Labs 2013] Sensio Labs. *Symfony website*, 2013. (Cited on page 28.)

[Lano 2009] Kevin Lano. Model-driven software development with uml and java. Course Technology, 2009. (Cited on page 19.)

[Lederberg 1987] J. Lederberg. *How DENDRAL was conceived and born.* In Proceedings of ACM conference on History of medical informatics, HMI '87, pages 5–19, New York, NY, USA, 1987. ACM. (Cited on page 17.)

[Lehman 2006] Christopher Wynn Lehman. *A rule-based expert system for the diagnosis of convergence problems in circuit simulation.* PhD thesis, 2006. AAI3209104. (Cited on page 17.)

[Li 2010] Tao Li, Wei Peng, C. Perng, Sheng Ma and Haixun Wang. *An Integrated Data-Driven Framework for Computing System Management.* Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, vol. 40, no. 1, pages 90 –99, jan. 2010. (Cited on page 18.)

[Littlestone 1994] Nick Littlestone and Manfred K. Warmuth. *The weighted majority algorithm*. Inf. Comput., vol. 108, no. 2, pages 212–261, February 1994. (Cited on page 31.)

[Liu 2007] Bing Liu. Web data mining: Exploring hyperlinks, contents, and usage data. Springer, 2007. (Cited on page 70.)

[Lomasky 2007] R. Lomasky, C. E. Brodley, M. Aernecke, D. Walt and M. Friedl. *Active Class Selection*. In Proceedings of the 18th European conference on Machine Learning, ECML '07, pages 640–647, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 31.)

[Lupu 1999] Emil C. Lupu and Morris Sloman. *Conflicts in Policy-Based Distributed Systems Management*. IEEE Trans. Softw. Eng., vol. 25, no. 6, pages 852–869, November 1999. (Cited on page 26.)

[M. Saar-Tsechansky 2009] P. Melville M. Saar-Tsechansky and F. Provost. *Active feature-value acquisition*. Management Science, vol. 55, pages 664–684, 2009. (Cited on page 31.)

[Magee 1989] Jeff Magee and Morris Sloman. *Constructing Distributed Systems in Conic*. IEEE Trans. Softw. Eng., vol. 15, no. 6, pages 663–675, June 1989. (Cited on page 25.)

[Mannila 1997] Heikki Mannila, Hannu Toivonen and A. Inkeri Verkamo. *Discovery of Frequent Episodes in Event Sequences*. Data Min. Knowl. Discov., vol. 1, no. 3, pages 259–289, January 1997. (Cited on page 70.)

[Martin 2007] P. Martin, W. Powley, K. Wilson, W. Tian, T. Xu and J. Zebedee. *The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services*. In Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 18.)

[Miller 2010] J Miller. *Design For Convention Over Configuration*. Microsoft MSDN Magazine, April 2010. (Cited on page 28.)

[Monteleoni 2006] Claire E. Monteleoni. *Learning with online constraints: shifting concepts and active learning*. Rapport technique, Massachusetts Institute of Technology, 2006. (Cited on page 31.)

[Naser Samy 2008] S. Abu Ola Abu Zaiter A Naser Samy. *An expert system for diagnosing eye diseases using CLIPS*. pages 923–924, 2008. (Cited on page 17.)

[Nierlein 2010] Sven Nierlein. *Mod_ gearman website*, 2010. (Cited on page 16.)

[Olsson 2008] Fredrik Olsson. *Bootstrapping Named Entity Annotation by Means of Active Machine Learning*. PhD thesis, University of Gothenburg, 2008. (Cited on page 31.)

[P. Melville 2004] F. Provost P. Melville M. Saar-Tsechansky and R. Mooney. *Active feature-value acquisition for classifier induction*. In In Proceedings of the IEEE Conference on Data Mining (ICDM), pages 483–486, 2004. (Cited on page 31.)

[PARC 2001] PARC. *AspectJ framework*, 2001. (Cited on page 19.)

[Parekh 2006] Janak Parekh, Gail Kaiser, Philip Gross and Giuseppe Valetto. *Retrofitting Autonomic Capabilities onto Legacy Systems*. Cluster Computing, vol. 9, no. 2, pages 141–159, April 2006. (Cited on page 25.)

[Pasquier 1999] Nicolas Pasquier, Yves Bastide, Rafik Taouil and Lotfi Lakhal. *Discovering Frequent Closed Itemsets for Association Rules*. In Proceedings of the 7th International Conference on Database Theory, ICDT '99, pages 398–416, London, UK, UK, 1999. Springer-Verlag. (Cited on page 70.)

[Patterson 1988] David A. Patterson, Garth Gibson and Randy H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*. SIGMOD Rec., vol. 17, no. 3, pages 109–116, June 1988. (Cited on page 2.)

[Pei 2000] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl and Hua Zhu. *Mining Access Patterns Efficiently from Web Logs*. In Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications, PADKK '00, pages 396–407, London, UK, UK, 2000. Springer-Verlag. (Cited on page 70.)

[Poirot 2008] P.-E. Poirot, J. Nogiec and Shangping Ren. *A Framework for Constructing Adaptive and Reconfigurable Systems*. Nuclear Science, IEEE Transactions on, vol. 55, no. 1, pages 284 –289, feb. 2008. (Cited on page 19.)

[P.R.J. Tillotsona 2004] P.M. Hughes P.R.J. Tillotsona Q.H. Wua. *Multi-agent learning for routing control within an Internet environment*. March 2004. (Cited on page 27.)

[Proffitt 2009] Brian Proffitt. *What Is Linux: An Overview of the Linux Operating System*, April 2009. (Cited on page 24.)

[Project 2013a] The Ganglia Project. *Ganglia website*, 2013. (Cited on page 99.)

[Project 2013b] The Icinga Project. *icinga website*, 2013. (Cited on pages 15 and 16.)

[Riedmiller 2009] Martin Riedmiller, Thomas Gabel, Roland Hafner and Sascha Lange. *Reinforcement learning for robot soccer*. Auton. Robots, vol. 27, no. 1, pages 55–73, July 2009. (Cited on page 27.)

[Roblee 2005] Christopher Roblee and George Cybenko. *Implementing Large-Scale Autonomic Server Monitoring Using Process Query Systems*. In Proceedings of the Second International Conference on Automatic Computing, ICAC '05, pages 123–133, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 25.)

[Rod Johnson 2009] Juergen Hoeller Rod Johnson and all. *Java Spring framework, Convention over Configuration*, 2009. (Cited on page 28.)

[Sadjadi 2005] S. Masoud Sadjadi, Philip K. McKinley and Betty H. C. Cheng. *Transparent shaping of existing software to support pervasive and autonomic computing*. SIGSOFT Softw. Eng. Notes, vol. 30, pages 1–7, May 2005. (Cited on page 19.)

[Sarma 2012] Shikhar Kr. Sarma, Kh. Robindro Singh and Abhijeet Singh. *knowledge acquisition of an expert system for rice plant disease diagnosis*. October 2012. (Cited on page 17.)

[Schmerl 2002] Bradley Schmerl and David Garlan. *Exploiting architectural design knowledge to support self-repairing systems*. In Proceedings of the 14th international conference on Software engineering and knowledge engineering, SEKE '02, pages 241–248, New York, NY, USA, 2002. ACM. (Cited on page 25.)

[Sheng 2006] Victor S. Sheng and Charles X. Ling. *Feature value acquisition in testing: a sequential batch test algorithm*. In Proceedings of the 23rd international conference on Machine learning, ICML '06, pages 809–816, New York, NY, USA, 2006. ACM. (Cited on page 31.)

[Shepler 2003] S. Shepler and all. *Network File System (NFS) version 4 Protocol*. Rapport technique, Network Working Group, 2003. (Cited on page 94.)

[SIA 2012] Zabbix SIA. *Zabbix website*, 2012. (Cited on page 15.)

[Sleator 1985] Daniel D. Sleator and Robert E. Tarjan. *Amortized efficiency of list update and paging rules*. Commun. ACM, vol. 28, no. 2, pages 202–208, February 1985. (Cited on page 31.)

[Solomon 2010] Bogdan Solomon, Dan Ionescu, Marin Litoiu and Gabriel Iszlai. *Autonomic computing control of composed web services*. In Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '10, pages 94–103, New York, NY, USA, 2010. ACM. (Cited on page 18.)

[Srikant 1995] Ramakrishnan Srikant and Rakesh Agrawal. *Mining Generalized Association Rules*. In Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95, pages 407–419, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. (Cited on page 70.)

[Srikant 1996a] Ramakrishnan Srikant and Rakesh Agrawal. *Mining quantitative association rules in large relational tables.* SIGMOD Rec., vol. 25, no. 2, pages 1–12, June 1996. (Cited on page 70.)

[Srikant 1996b] Ramakrishnan Srikant and Rakesh Agrawal. *Mining Sequential Patterns: Generalizations and Performance Improvements.* In Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '96, pages 3–17, London, UK, UK, 1996. Springer-Verlag. (Cited on page 70.)

[Sterritt 2005] Roy Sterritt, Barry Smyth and Martin Bradley. *PACT: Personal Autonomic Computing Tools.* In Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, ECBS '05, pages 519–527, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 25.)

[Stoilov 2010] Todor Stoilov and Krasimira Stoilova. *Potential formal models for autonomic computing applications.* In Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies, CompSysTech '10, pages 305–310, New York, NY, USA, 2010. ACM. (Cited on page 19.)

[Sutton 1998] Richard S. Sutton and Andrew G. Barto. Introduction to reinforcement learning. MIT Press, Cambridge, MA, USA, 1st édition, 1998. (Cited on page 27.)

[Tarjan 1971] Robert Tarjan. *Depth-First Search and Linear Graph Algorithms.* In SIAM Journal on Computing, pages 146–160, August 1971. (Cited on page 47.)

[team 2010] Shinken team. *Shinken website*, 2010. (Cited on page 15.)

[Team 2013] Rails Core Team. *Ruby on rails website*, 2013. (Cited on page 28.)

[TIA 2010] *Data Center Standards Overview.* White Paper (http://www.adc.com/us/en/Library/Literature/102264AE.pdf), 2010. (Cited on page 4.)

[Tong 2001] Simon Tong. *Active Learning: Theory and Applications.* PhD thesis, Standford University, 2001. (Cited on page 31.)

[Tong 2002] Simon Tong and Daphne Koller. *Support vector machine active learning with applications to text classification.* J. Mach. Learn. Res., vol. 2, pages 45–66, March 2002. (Cited on page 31.)

[Trencansky 2006] Ivan Trencansky, Radovan Cervenka and Dominic Greenwood. *Applying a UML-based agent modeling language to the autonomic computing*

domain. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, pages 521–529, New York, NY, USA, 2006. ACM. (Cited on page 19.)

[Vernon Turner 2005] IDC Vernon Turner. Defining the landscape: Trends and forecasts for the enterprise server market and data centers. Sun Summit on 21st Century Eco-Responsibility, 2005. (Cited on page 3.)

[Vidal 2003] J M Vidal. *Learning in Multiagent Systems: An Introduction from a Game-Theoretic Perspective*. Rapport technique cs.MA/0308030, Aug 2003. (Cited on page 27.)

[Vovk 1990] Volodimir G. Vovk. *Aggregating strategies*. In Proceedings of the third annual workshop on Computational learning theory, COLT '90, pages 371–386, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. (Cited on page 31.)

[W. Strickland 2005] Jonathan W. Strickland, Vincent W. Freeh and Sudharshan S. Vazhkudai. *Governor: Autonomic Throttling for Aggressive Idle Resource Scavenging*. In Proceedings of the Second International Conference on Automatic Computing, ICAC '05, pages 64–75, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 25.)

[X. Chai 2004] Q. Yang X. Chai L. Deng and Charles X. Ling. *Test-Cost Sensitive Naive Bayes Classification*. In Proceedings of the Fourth IEEE International Conference on Data Mining, ICDM '04, pages 51–58, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 31.)

[Xu 2005] Jing Xu and Jose A. B. Fortes. *Towards Autonomic Virtual Applications in the In-VIGO System*. In Proceedings of the Second International Conference on Automatic Computing, ICAC '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 25.)

[Y. 2002] Bigus J. P.; Schlosnagle D. A.; Pilgrim J. R.; Mills III W. N.; Diao Y. *ABLE: A toolkit for building multiagent autonomic systems*. IBM Systems Journal, vol. 41, pages 350–371, 2002. (Cited on pages 18 and 25.)

[Zaki 1997] Mohammed J Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara and Wei Li. *New Algorithms for Fast Discovery of Association Rules*. Rapport technique, Rochester, NY, USA, 1997. (Cited on page 70.)

[Zaki 2000] Mohammed J. Zaki. *Scalable Algorithms for Association Mining*. IEEE Trans. on Knowl. and Data Eng., vol. 12, no. 3, pages 372–390, May 2000. (Cited on page 71.)

[Zheng 2002] Z. Zheng and B. Padmanabhan. *On active learning for data acquisition*. In In Proceedings of the IEEE Conference on Data Mining (ICDM), pages 562–569, 2002. (Cited on page 31.)

[Zheng 2004] Tong Zheng. *WebFrame, In pursuit of computationally and cognitively efficient web mining.* PhD thesis, University of Alberta, 2004. (Cited on page 70.)

[Zhu 2005] Xiaojin Zhu. *Semi-Supervised Learning with Graphs.* PhD thesis, Carnegie Mellon University, 2005. (Cited on page 31.)

Appendix A

Database

The database contains 19 tables :

- MetaAgent: this the table in which all the MetaAgent are defined. The fields of this table are enough to fully define Coordinators, which thus do not need an extra table.
- Veto: associates the Coordinator to their Veto
- FileAgent: this table completes information from the MetaAgent table for the FileAgents and the FolderAgent.
- FileContentRules: contains the patterns that files defined in the FileAgent table are supposed to match.
- FilenameFilter: it contains the patterns that the content of a Folder has to match in order to be taken into account.
- ProcessAgent: this table completes information from the MetaAgent table for the ProcessAgent.
- Limits: contains the default limits that the system puts on processes.
- ProcessLimits: contains the custom limits for a process defined in the ProcessAgent table.
- EnvironmentAgent: this table completes information from the MetaAgent table for the EnvironmentAgent.
- IgnoredFS: list of filesystems to ignore, linked to a given EnvironmentAgent.
- Server: contains the definition of the Servers
- MetaAgentTree: defines which MetaAgent is a child of which Coordinator
- Rule: the dependency rules
- RecoveryRule: the recovery rules
- Total and Occurrence: used to store information for reinforcement learning.
- RecoveryHistory, recoveryHistory_Problem, recoveryHistory_Action: define all the diagnoses and recoveries' history. In particular, what were the problems before and after what action on each Agent.

The full schema is visible in Fig. A.1

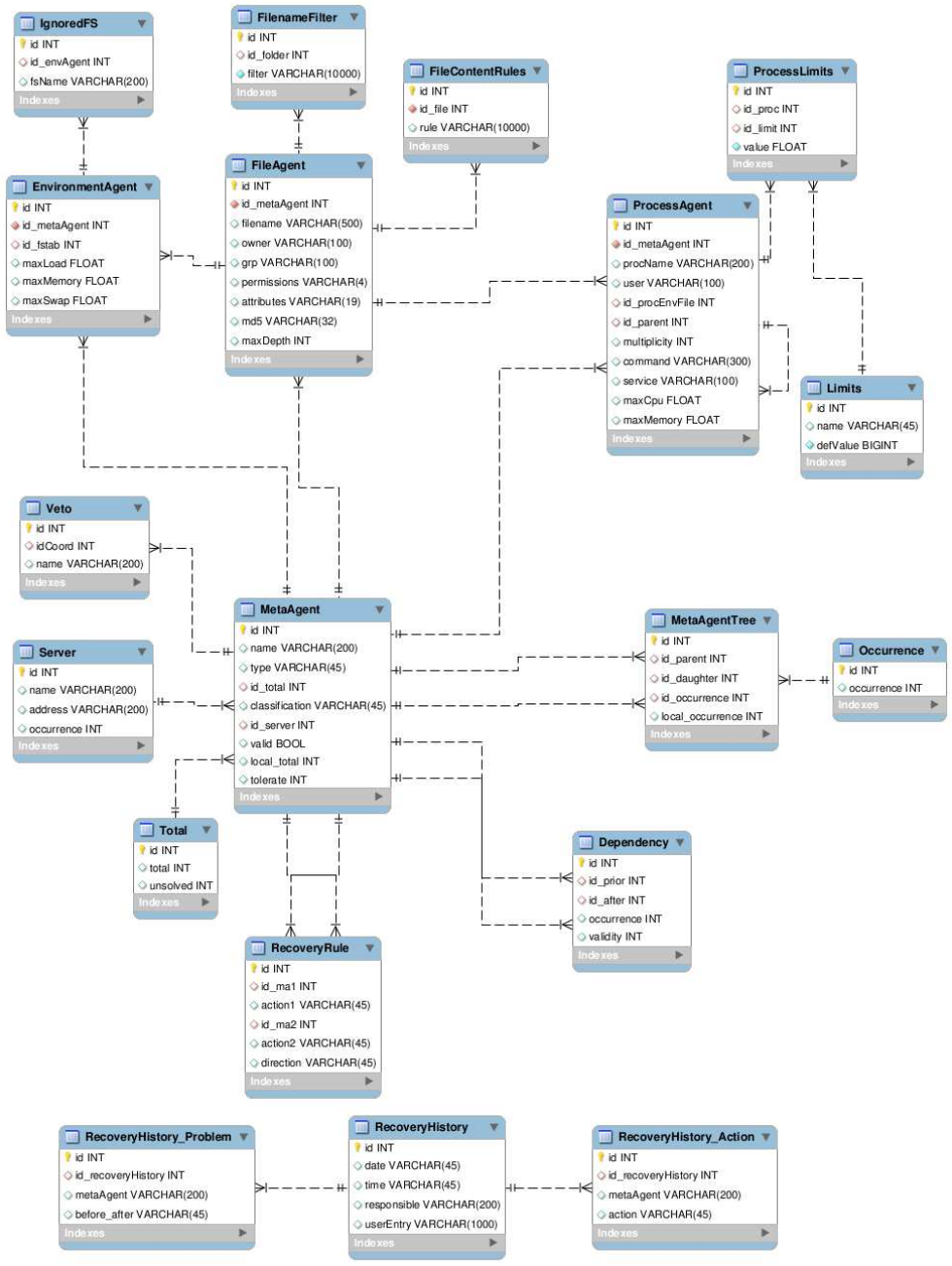


Figure A.1: Complete schema of the database

Appendix B

Phronesis grammar

B.1 Formal description of the grammar

Following the formalism described my Noam Chomsky in [Chomsky 1957] and [Chomsky 1956], one can define the Phronesis grammar using:

- A finite set N of nonterminal symbols
- A finite set Σ of terminal symbols
- A finite set P of production rules
- A distinguished symbol $S \in N$ that is the start symbol.

In the case of the Phronesis Grammar, Σ is composed of the following elements: *{Coordinator, set, veto, needs, FileAgent, FolderAgent, ProcessAgent, requires, tolerate , #, before, after, trigger, on, triggerSuffix, content, nameFilter, parent, mountpoint, maxLoad, maxMemory, maxSwap, setEnv, Server, attach, detach, import}*.

The nonterminal symbols set N is *{agent_type, attribute, coordinatorDef, coordinator_end, coordinatorMiddle, coordinator_needs, coordinator_needs_assign, coordinator_requires, coordinator_start, coordinator_tolerate, coordinator_veto, env_constraint, file_content, fileDef, file_end, file_middle, file_start, folderDef, folder_end, folder_filenameFilter, folder_middle, folder_start, importStatment, load_env, memory_env, metaAgent_attribute, metaAgent_end, metaAgent_inheritance, metaAgent_middle, metaAgent_setTriggerSuffix, metaAgent_start, metaAgent_trigger, mountpointDefinition, mountpoint_env, name, number, parent_attribute, processDef, process_end, process_midle, process_start, server_attachCoord, serverDef, server_detachCoord, server_end, server_inheritance, server_middle, server_start, setEnvDef, Sigma, swap_env, variableAssignment, variable_name, variable_value, when, definitions, phr_config_file}*.

The production rules are:

- `agent_type = Coordinator`
- `agent_type = FileAgent`
- `agent_type = FolderAgent`
- `agent_type = ProcessAgent`

- name = Word(alphanums + '_')
- number = Word(nums)
- attribute = Word(alphanum $/*_ -.=\\^[]())
- variable_name = "$" name
- variable_value = attribute
- variable_value = variable_value variable_value
- variableAssignment = "set" variable_name variable_value ";"
- when = "before"
- when = "after"
- coordinator_inheritance -> ":" name
- coordinator_start = "Coordinator" name "{"
- coordinator_start = "Coordinator" name coordinator_inheritance "{"
- coordinator_needs_assign = name "=" name
- coordinator_needs_assign = name "=" variable_name
- coordinator_veto = "veto" name ";"
- coordinator_needs = "needs" agent_type name ";"
- coordinator_needs = "needs" agent_type name "=" name ";"
- coordinator_requires = "requires" name ";"
- coordinator_tolerate = "tolerate" number ";"
- coordinator_end = "}"
- coordinatorMiddle = coordinator_tolerate
- coordinatorMiddle = coordinator_veto
- coordinatorMiddle = coordinator_needs
- coordinatorMiddle = coordinator_needs_assign
- coordinatorMiddle = variableAssignment
- coordinatorMiddle = coordinator_requires
- coordinatorMiddle = coordinatorMiddle coordinatorMiddle
- coordinatorDef = coordinator_start coordinator_end
- coordinatorDef = coordinator_start coordinatorMiddle coordinator_end
- metaAgent_inheritance = ":" name
- metaAgent_start = name "{"
- metaAgent_start = name metaAgent_inheritance "{"
- metaAgent_attribute = name "->" attribute ";"
- metaAgent_trigger = when name "trigger" name "on" name ";"
- metaAgent_setTriggerSuffix = "triggerSuffix" "->" variable_name ";"
- metaAgent_middle = metaAgent_setTriggerSuffix
- metaAgent_middle = metaAgent_attribute
- metaAgent_middle = variableAssignment
- metaAgent_middle = metaAgent_trigger

- metaAgent_middle = metaAgent_middle metaAgent_middle
- metaAgent_end = "}"
- file_start = "FileAgent" metaAgent_start
- file_content = "content" "->" attribute ";"
- file_end = metaAgent_end
- file_middle = file_content
- file_middle = metaAgent_middle
- file_middle = file_middle file_middle
- fileDef = file_start file_end
- fileDef = file_start file_middle file_end
- folder_start = "FolderAgent" metaAgent_start
- folder_end = "}"
- folder_filenameFilter = "nameFilter" "->" attribute ";"
- folder_middle = folder_filenameFilter
- folder_middle = file_middle
- folder_middle = folder_middle folder_middle
- folderDef = folder_start folder_end
- folderDef = folder_start folder_middle folder_end
- process_start = "ProcessAgent" metaAgent_start
- parent_attribute = "parent" "->" name ";"
- process_midle = metaAgent_middle
- process_end = metaAgent_end
- processDef = process_start   process_end
- processDef = process_start parent_attribute process_end
- processDef = process_start parent_attributeprocess_midle process_end
- processDef = process_start process_midle process_end
- mountpointDefinition = attribute
- mountpoint_env = "mountpoint" mountpointDefinition ";"
- load_env = "maxLoad" number ";"
- memory_env = "maxMemory" number ";"
- swap_env = "maxSwap" number ";"
- env_constraint = mountpoint_env
- env_constraint = load_env
- env_constraint = memory_env
- env_constraint = swap_env
- env_constraint = env_constraint env_constraint
- setEnvDef = "setEnv" name env_constraint
- server_inheritance = ":" name
- server_start = "Server" name "{"

- server_start = "Server" name server_inheritance "{"
- server_attachCoord = "attach" name ";"
- server_detachCoord = "detach" name ";"
- server_end = "}"
- server_middle = general_server_attribute
- server_middle = server_attachCoord
- server_middle = server_detachCoord
- server_middle = metaAgent_attribute
- server_middle = server_middle server_middle
- serverDef = server_start server_end
- serverDef = server_start server_middle server_end
- path = Word(alphanums ./_)
- importStatment = "import" path ";"
- definitions = coordinatorDef
- definitions = fileDef
- definitions = folderDef
- definitions = processDef
- definitions = serverDef
- definitions = definition definition
- phr_config_file = importStatment phr_config_file
- phr_config_file = definitions

And finally, the Start symbol is *"phr_ config_ file"*.

B.2 Examples of Phronesis configurations

B.2.1 Example 1 : websites

This configuration demonstrates how one can define abstract MetaAgents for a website, and then inherit them to instantiate real websites. The comments inside the configuration file detail each action and give information on the consequences of them.

```
# We start by defining a completely abstract website
# which is made of an httpd process (with one child),
# and 2 coordinators : one for the data and one for the
# configuration


# This ProcessAgent is the definition of a standard
# httpd root process : the binary is /usr/sbin/httpd
# and should be run as root. It is started with
# a standard linux process called "httpd". There should only
# be 1 instance of this process.
# Eventhough this is not very standard, we have defined
# two constraints on our process : it should have an
# environment variable ORACLE_LIB whose value is
# /usr/lib/oracle/, and the maximum number of file
# descriptors is 2000.
# The server on which it runs is a variable called $webSrv
# to be defined later
```

```
ProcessAgent abs_httpd {
    procName -> /usr/sbin/httpd;
    multiplicity -> 1;
    user -> root;
    service -> httpd;
    server -> $webSrv;
    envVar -> ORACLE_LIB=/usr/lib/oracle/;
    open_files -> 2000;
}

# The httpd process has one root process (abs_httd)
# and then forks off a lot. This ProcessAgent describes
# the children. Because we define abs_httpd as the parent
# process, abs_httpdChild will have all the attributes
# of its parent (procName, multiplicity, user, service,
# server envVar and open_files). However, we overwrite
# two of those attributes : the multiplicity
# (a maximum number of children processes can actually
# be set in the real httpd configuration files)
# and the user, which is set to a variable $webUser
ProcessAgent abs_httdpChild {
    parent -> abs_httpd;
    multiplicity -> 100;
    user -> $webUser;
}

# We define a Coordinator to represents the webSite.
# It has three needs, out of which only one is assigned
# to abs_httpd.
Coordinator abs_webSite {
    needs Coordinator n_c_data;
    needs Coordinator n_c_conf;
    needs ProcessAgent n_p_httpd = abs_httpd;
}

# We can now define a default website

# The Coordinator where the data are stored
# is composed by only one folder.

FolderAgent default_documentRoot {
    filename -> /var/www/html;
    server -> $webSrv;
}

Coordinator default_Data {
    needs FolderAgent n_fo_data = default_documentRoot;
}

# The Coordinator that contains the configuration
# is composed by only one file

# We expect the file to contain a line with "DocumentRoot"
# as well as a line containing "Listen 80"
FileAgent default_httpdConf {
    filename -> /etc/httpd/conf/httpd.conf;
    owner -> root;
    grp -> root;
    content -> Listen 80;
    content -> DocumentRoot;
    server -> $webSrv;
}

Coordinator default_Conf {
    needs FileAgent n_fi_conf = default_httpdConf;
}


# default_website inherits from abs_webSite.
# This means that default_webSite has 3 needs :
#  - one ProcessAgent, which is already assigned
#    to a copy of the abs_httpChild
#  - two Coordinators, non assigned yet
# We assign the Coordinators previously defined
# to our two needs.
# We assign a value to the $webUser variable :
# this assignation is propagated in all the tree,
# so it will reach the copy of abs_httpdChild.
Coordinator default_webSite : abs_webSite {
    n_c_data = default_Data;
    n_c_conf = default_Conf;
    set $webUser apache;
}
```

```
# We need to define servers on which to run

# First, we define an abstract Server, a model
# for all the webserver :
# It will have 2 montpoints, and two file system have
# to be ignored. Also, we override the default maxLoad
# to a value of 9.

Server abs_srv {
    mountpoint -> nfs:/nfsexports/sharedFiles /sharedFiles nfs rw,hard,intr,nolock 0 0;
    mountpoint -> nfs:/nfsexports/otherFiles /otherFiles nfs rw,hard,intr 0 0;
    ignoreFs -> /castorfs;
    ignoreFs -> /cvmfs;
    maxLoad -> 9;
}


# Because srv1 inherits from abs_server,
# it will have all the caracteristic from abs_server,
# and its address is 192.168.0.50
Server srv1 : abs_srv {
    address -> 192.168.0.50;
}

# Because srv2 inherits from abs_server,
# it will have all the caracteristic from abs_server,
# except the maxLoad which is redefined to 6.
# Its address is 192.168.0.51
Server srv2 : abs_srv{
    address -> 192.168.0.51;
    maxLoad -> 6;
}


# Site1 inherits from default_website
# This will create 2 Coordinators :
#   - one containing a newly created FolderAgent, for the data
#   - one containing a newly created FileAgent, for the configuration
# It will also create 2 ProcessAgents to represent the http process.
# The only unknown left is the server on which to run, which is assigned
# with a simple line. (The value of $webUser is apache since Site1
# inherits from default_webSite)
Coordinator Site1 : default_webSite {
    set $webSrv srv1;
}

# As for Site1
Coordinator Site2 : default_webSite {
    set $webSrv srv2;
}
```

It can be seen in Fig. B.1 that all the sites share the occurrences id 1,2, and 3 that they inherit from abs_website, and that the occurrences id 4 and 5 are inherited from default_website and shared by Site1 and Site2.

B.2.2 Example 2 : servers

This second example illustrates how one can define a hierarchy of servers, attach and detach Coordinators.

```
# We define two Coordinators that will be attached to Servers

# Note that eventhoug mandatory, the
# server attribute is not set. It is because
# it will be set when attached to a Server

FileAgent conf1 {
  filename -> /path/to/conf;
}

FileAgent conf1bis {
  filename -> /path/to/conf/bis;
}

# The first Coordinator contains 2 FileAgents
# However, the "tolerate 1" directive means
```

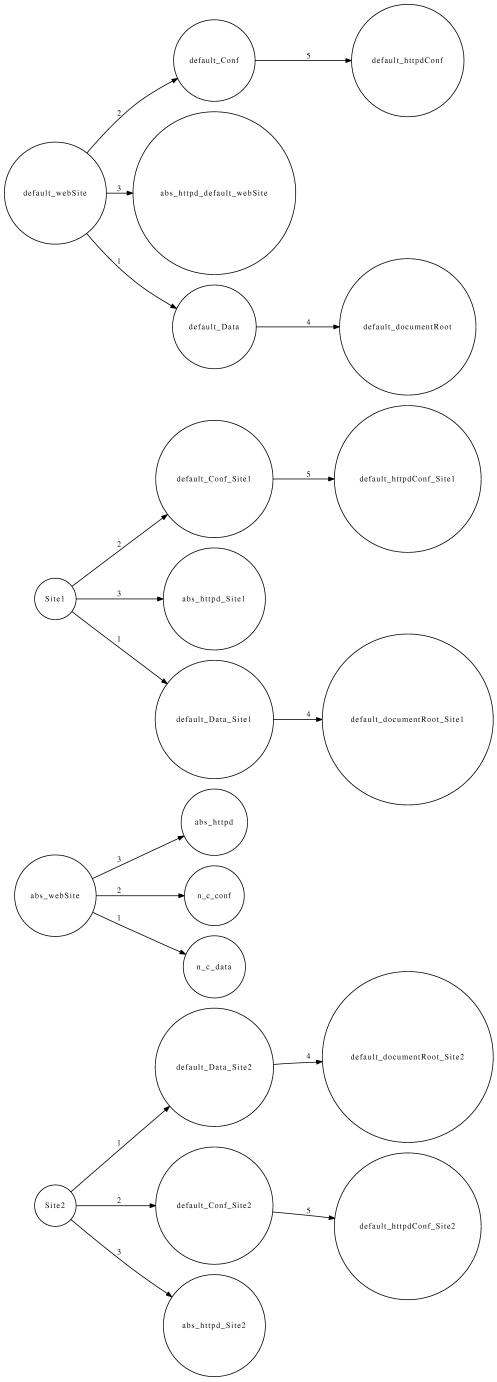


Figure B.1: Representation of the configuration of an abstract website and two instances

```
18 # that out of the two, only one needs to be okay.
   # This is useful for example when a file used for
20 # the same purpose (e.g. ldap configuration)
   # is in a different location
22 # depending on the operating system version.

24 Coordinator attach1 {
     tolerate 1;
26   needs FileAgent f1 = conf1;
     needs FileAgent f1bis = conf1bis;
28 }

30
   # On this file, the server attribute is assigned
32 # a variable. When attached to a Server, the compiler
   # will make sure to remove the server from the list
34 # of attributes waiting for $variable to be set.

36 FileAgent conf2{
     filename -> /path/to/conf2/;
38   server -> $variable;
   }
40
   Coordinator attach2{
42   needs FileAgent f2 = conf2;
   }
44
   # abs_srv is an abstract server (it has no address)
46 # with two attached Coordinators.

48 Server abs_srv {
     attach attach1;
50   attach attach2;
   }
52
   # real_srv_1 inherits from abs_srv
54 # and so has two attached Coordinators
   # which are copies of attach1 and attach2
56 Server real_srv_1 : abs_srv {
     address -> 192.168.1.50;
58 }

60
   # real_srv_2 inherits from abs_srv
62 # but we detach attach1, so the only
   # attached Coordinator it has is
64 # a copy of attach2
   Server real_srv_2 : abs_srv {
66   detach attach1;
     address -> 192.168.1.51;
68 }
```

Appendix C

Tree comparison algorithm

This appendix will illustrate the algorithm which is used in order to update the configuration in the database without loosing all the experience already gathered. It is recommended to read appendix A first to learn about the database schema.

The content of the database at the beginning is visible in Fig. C.1 and was defined using C.1 . The labels on the verticies are the Occurrence Ids and the number in the edges are the Total Ids. Note that only the Coordinators have a Total Id.

Since Site1 and Site2 inherit from MetaSite, their Occurrence and Total Ids are the same. All other Ids are used only once.

Listing C.1: Configuration file originaly loaded in the database

```
Server localhost {
    address -> localhost;
}


ProcessAgent MetaHttpd {
        procName -> /usr/sbin/httpd;
        user -> root;
        server -> $webSrv;
}

Coordinator MetaSite {
        needs Coordinator MetaData;
        needs Coordinator MetaConf;
        needs ProcessAgent Metahttpd = MetaHttpd;
}

FileAgent F1 {
    server -> $webSrv;
    filename -> f1;
}

FileAgent F2 {
    server -> $webSrv;
    filename -> f2;
}


Coordinator Conf1 {
    needs FileAgent f = F1;
}

Coordinator Data1 {
    needs FileAgent f = F2;
}

Coordinator Site1 : MetaSite {
    MetaData = Data1;
    MetaConf = Conf1;
    set $webSrv localhost;
    set $webUser apache;
}
```

```
44
FileAgent F3 {
46      server -> $webSrv;
        filename -> f3;
48  }
50  FileAgent F4 {
        server -> $webSrv;
52      filename -> f4;
    }
54
    Coordinator Conf2 {
56      needs FileAgent f = F3;
    }
58
    Coordinator Data2 {
60      needs FileAgent f = F4;
    }
62
    Coordinator Site2 : MetaSite {
64      MetaData = Data2;
        MetaConf = Conf2;
66      set $webSrv localhost;
        set $webUser apache;
68  }
70  FolderAgent Y{
        filename -> Y;
72      server -> localhost;
    }
74
    Coordinator X{
76      needs FolderAgent n_x = Y;
    }
78
80  FileAgent D{
        filename -> D;
82      server -> localhost;
    }
84
    Coordinator B{
86      needs FileAgent f1 = D;
    }
88
    FileAgent E{
90      filename -> E;
        server -> localhost;
92  }
94  FileAgent F{
        filename -> F;
96      server -> localhost;
    }
98
    Coordinator C{
100     needs FileAgent f1 = E;
        needs FileAgent f2 = F;
102 }
104
    Coordinator A{
106     needs Coordinator f1 = B;
        needs Coordinator f2 = C;
108 }
```

The new configuration file visible in C.2 has some differences with the previous one:

- The tree composed by X and Y is removed.
- A new tree composed by U and V appears.
- The complete tree of Site2 is removed.
- A new tree, Site3, inherits from MetaSite.
- Part of the A tree is removed : A itself, as well as the complete B tree.

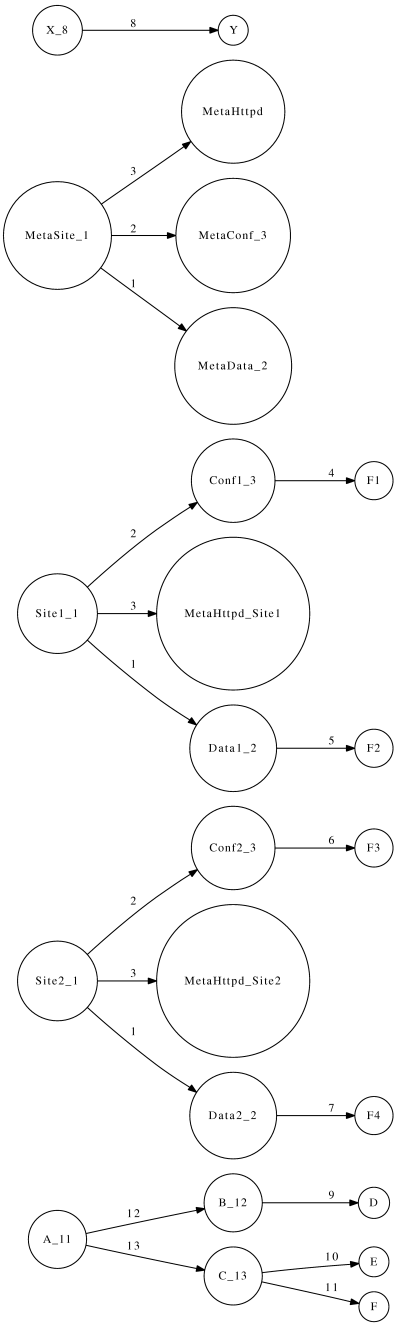


Figure C.1: Content of the database

- The Agent F3 that was originaly part of Conf2 is now part of Data1.
- The Agent F4 that was originaly part of Data2 is now part of Conf3.

Listing C.2: New configuration file to update the database

```
Server localhost {
    address -> localhost;
}

FolderAgent V{
    filename -> V;
    server -> localhost;
}

Coordinator U{
    needs FolderAgent n_v = V;
}

ProcessAgent MetaHttpd {
        procName -> /usr/sbin/httpd;
        user -> root;
        server -> $webSrv;
}

Coordinator MetaSite {
        needs Coordinator MetaData;
        needs Coordinator MetaConf;
        needs ProcessAgent Metahttpd = MetaHttpd;
}

FileAgent F1 {
    server -> $webSrv;
    filename -> f1;
}

FileAgent F2 {
    server -> $webSrv;
    filename -> f2;
    after ANY trigger FILE_CHOWN on F1;

}

FileAgent F3 {
    server -> $webSrv;
    filename -> f3;
}

Coordinator Conf1 {
    needs FileAgent f = F1;
}

Coordinator Data1 {
    needs FileAgent f = F2;
    needs FileAgent f3 = F3;
}

Coordinator Site1 : MetaSite {
    MetaData = Data1;
    MetaConf = Conf1;
    set $webSrv localhost;
    set $webUser apache;
}


FileAgent F4 {
    server -> $webSrv;
    filename -> f4;
}

FileAgent F5 {
    server -> $webSrv;
    filename -> f5;
}

Coordinator Conf3 {
    needs FileAgent f = F4;
}

Coordinator Data3 {
    needs FileAgent f = F5;
```

```
76  }
78  Coordinator Site3 : MetaSite {
        MetaData = Data3;
80      MetaConf = Conf3;
        set $webSrv localhost;
82      set $webUser apache;
    }
84
    FileAgent E{
86      filename -> E;
        server -> localhost;
88  }
90  FileAgent F{
        filename -> F;
92      server -> localhost;
    }
94
    Coordinator C{
96      needs FileAgent f1 = E;
        needs FileAgent f2 = F;
98  }
100 Coordinator G{
        requires C;
102 }
```

Because one wants to keep the experience already gathered, the update algorithm must ensure that removing Site2 will not destroy all the experience about the websites, and that removing part of the A tree will not destroy the experience of the C tree, since it remains valid.

The first step consists of a normal parsing of the file, without looking at what is in the database. This parsing produces objects represented in Fig. C.2

The role of the update algorithm is to find what has changed based on the criteria described in section IV.2.1.iii. For each of the Occurrences and the Total Ids, the algorithm will establish three lists:

- Elements to remove: some indexes are not needed anymore and can be removed.
- Elements to create: when new independent entities are created, they need to be assigned a unique index.
- Elements to update: entities that are already present in the database must conserve their index, so the algorithm establishes a correspondence between the index assigned during this compilation and the one set in the database.

In this example, the Total ids lists produced by the algorithm will be the following:

Total Ids to delete:

- 8: Coordinator X
- 11: Coordinator A
- 12: Coordinator B

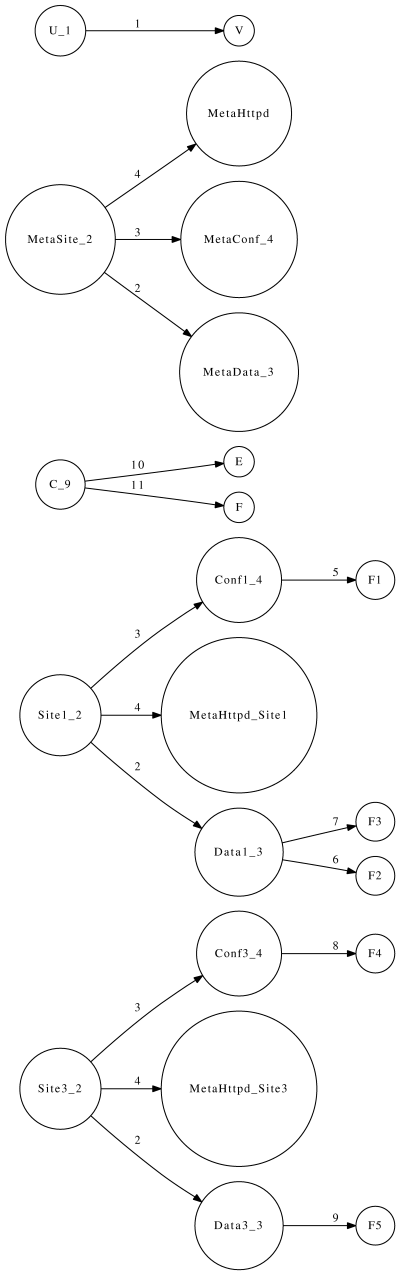


Figure C.2: Content of the new configuration

Thee three Coordinators do not exist in the new configuration anymore and can thus be deleted. However, even if Site2 is deleted, the total Id 1 associated with it is not removed, because it is used by all the other websites.

Total Ids to create:

- 15: Coordinator U is completely new and thus needs to be assigned a Total Id which is not yet used in the database. This is simply done by attributing indexes starting from the highest one in the database.

The algorithm also constructs a mapping list to match the new indexes to the old:

- 1 ⇒ 15: Coordinator U that was assigned 1 during the parsing will have 15 in the database.
- 2 ⇒ 1: The Total id used for the MetaSite Coordinators was assigned 2 during the parsing, and is matched to 1 to be the same than what is already in the database.
- 3 ⇒ 2: The Total id used for the Data Coordinators is matched to 2.
- 4 ⇒ 3: The Total id used for the Conf Coordinators is matched to 3.
- 9 ⇒ 13: The Total id of C is matched from 9 to 13.

As for the Total ids, one can establish by looking at Fig. C.1 and Fig. C.2 what lists the algorithm will produce.

Occurrence Ids to delete:

- 5 (Data1 ⇒ F2): The tree bellow Data1 is changed, so the experience there is not valid anymore.
- 6 (Conf2 ⇒ F3): Site2 is removed and this Occurrence id was not shared.
- 7 (Data2 ⇒ F4): Site2 is removed and this Occurrence id was not shared.
- 8 (X ⇒ Y): The X tree is removed.
- 9 (B ⇒ D): The B tree is removed.
- 12 (A ⇒ B): The B tree is removed.
- 13 (A ⇒ C): The A Coordinator is removed.

New Occurrence Ids are created:

- 14: U ⇒ V
- 15: Data1 ⇒ F3

- 16: Data1 ⇒ F2
- 17: Data3 ⇒ F5
- 18: Conf3 ⇒ F4

Note that the Agents F3 and F4 will remain in the database and will not be newly created; however all their experience is reinitialized.

Finally, as for the Total Ids, a correspondence is established between the newly parsed Ids and those already in the database:

- 1 ⇒ 17: Occurrence id for Data3 ⇒ F5
- 2 ⇒ 1: Occurrence id for MetaSite ⇒ MetaData, shared by all the sites
- 3 ⇒ 2: Occurrence id for MetaSite ⇒ MetaConf, shared by all the sites
- 4 ⇒ 3: Occurrence id for MetaSite ⇒ MetaHttpd, shared by all the sites
- 5 ⇒ 4: Occurrence id for Conf1 ⇒ F1
- 6 ⇒ 14: Occurrence id for U ⇒ V
- 7 ⇒ 15: Occurrence id for Data1 ⇒ F3
- 8 ⇒ 18: Occurrence id for Conf3 ⇒ F4
- 9 ⇒ 16: Occurrence id for Data1 ⇒ F2

Similar lists are established for all the other Ids used in the database, but they do not suffer the same complications since they are all unique: either the entity to which they are attached is kept or not, and there is no need to make sure that the id is used by another entity.

Appendix D

API usage

This appendix shows an example of a class that inherits from the AbstractPhronesisClient class. It is necessary to inherit from it and to implement the pure virtual *"update"* method. The class phrExampleInterface defined in D.1 implements this method by printing the messages it receives using different colors (red for errors, blue for questions and so on).

Listing D.1: Example of API inheritance class

```
#include <iostream>
#include <fstream>
#include "phronesisAnalyzerClientApi.hpp"


#define START_COLOR "\033[1;"
#define START_RED START_COLOR"31m"
#define START_GREEN START_COLOR"32m"
#define START_YELLOW START_COLOR"33m"
#define START_BLUE START_COLOR"34m"
#define END_COLOR "\033[0m"

class phrExampleInterface: public AbstractPhronesisClient {
public:

    phrExampleInterface() :
            AbstractPhronesisClient() {
    }

    phrExampleInterface(std::string serverAddr, std::string port) :
            AbstractPhronesisClient(serverAddr, port) {
    }

    void update(api_message e) {
        switch (e.m_type) {
        case api_message::NEW_MSG: {
            std::cout << START_YELLOW << "[NEW_MESSAGE] (" << e.m_id
                    << ") : " << e.m_text << END_COLOR;
            break;
        }
        case api_message::NEW_LOG: {
            std::cout << "[NEW LOG] " << e.m_text;
            break;
        }
        case api_message::NEW_QUESTION: {
            std::cout << START_BLUE << "[NEW QUESTION] (" << e.m_id
                    << ")    : " << e.m_text << END_COLOR;
            break;
        }
        case api_message::NEW_YN_QUESTION: {
            std::cout << START_BLUE << "[YES/NO] (" << e.m_id << ") : "
                    << e.m_text << END_COLOR;
            break;
        }
        case api_message::NEW_PB_QUESTION: {
            std::cout << START_BLUE << "[PROBLEM] " << e.m_text
                    << END_COLOR;
            break;
        }
        case api_message::ERROR_TEXT: {
            std::cout << START_RED << "[ERROR] : " << e.m_text << END_COLOR;
            break;
        }
        case api_message::STATISTIC: {
            std::cout << "[STATISTIC] : " << e.m_text << " " << e.m_id;
```

```
            break;
        }
        case api_message::CURRENT_PB: {
            std::cout << START_GREEN << "[CURRENT PROBLEM] : " << e.m_text
                    << END_COLOR;
            break;
        }
        case api_message::NEW_VETO: {
            std::cout << "[VETO] " << e.m_text << " : " << e.m_id
                    << std::endl;
            break;
        }
        default:
            std::cout << "[UNKNOWN TYPE] " << e;
            break;
        }
        std::cout << std::endl;

    }
}
;
```

The code in D.2 shows how one can simply use the phrExampleInterface class to observe all what the Core sends as output communication. The program will never exit.

Listing D.2: Example of usage of the API for output communication

```
int main(int argc, char ** argv) {
    // define the server address and port
    std::string server = "127.0.0.1";
    std::string port = "7172";

    // create the class instance and give the server and the port to the constructor
    phrExampleInterface phrApi(server, port);

    // defaultRun can be used instead of run(server,port) since the arguments were
        provided to the constructor already
    phrApi.defaultRun();
    return 0;
}
```

The code in D.3 shows how one can simply use the phrExampleInterface class to send data to the Core: it periodically reads a file and send its content to the Core. One could imagine that this file is a regular dump of the monitoring status written by an external program. The program will exit only if there is a problem with the file read. As for the example D.2, all the output communication will be printed in the console, since the *"run"* method was started in a new thread.

Listing D.3: Example of usage of the API for input communication

```
int main(int argc, char ** argv) {
    std::string server = "127.0.0.1";
    std::string port = "7172";
    std::string line;
    bool error = false;
    phrExampleInterface phrApi;

    // Start the run method in a new thread.
    // The arguments server and port are needed because not provided
    // to the constructor
    boost::thread t(&phrExampleInterface::run, &phrApi, server,port);


    // Every 10 seconds, open the file and if there is a content, send it to the core.
    // Loop exists if there is a problem with the file
    while (!error) {
        std::ifstream myfile("errorFile.txt");
        if (myfile.is_open()) {
            getline(myfile, line);
            if (!line.empty())
```

```
                phrApi.giveProblemList(line);
            myfile.close();
        }
        else {
            std::cout << "Unable to open file";
            error = true;
        }
        sleep(10);
    }
    return error;
}
```

In order to compile, these examples must be linked against the boost_system, boost_thread and boost_serialization libraries.

Appendix E

UML diagrams

This appendix contains only a few UML class and interaction diagrams. The aim is to give a very brief overview of the Core software structure. For complete documentation, please refer to the doxygen generated pages.

The class diagrams visible in Fig. E.1, Fig. E.2, Fig. E.3 illustrate the object approach that was used as well as the will to factorize as much as possible the classes.

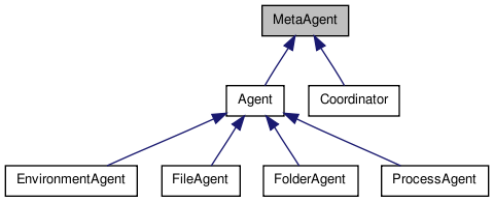


Figure E.1: MetaAgent UML class diagram

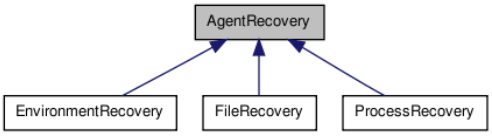


Figure E.2: AgentRecovery UML class diagram

When the Core receives a message from a user, it will interpret this message, and according to the message type will take actions, before eventually answering. Those actions involve several components of the Core, such as the VetoManager or the InteractionManager. The Fig. E.4 is an interaction diagram of what happens when a message is received.

The Syslog and the Cout observers are different from the API clients because they are local: they listen directly to the update of the InteractionManager without going through all the API network session mechanism. The Fig. E.5 shows the direct connections of the Syslog observer.

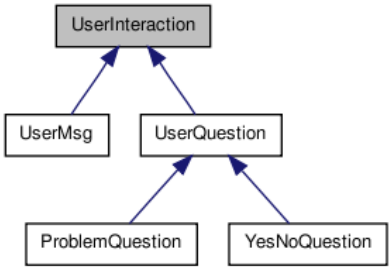


Figure E.3: UserInteraction UML class diagram

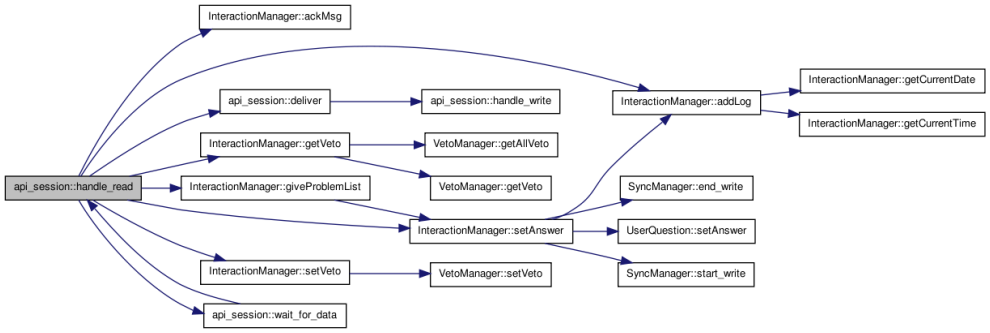


Figure E.4: Interaction diagram when the Core receives a message

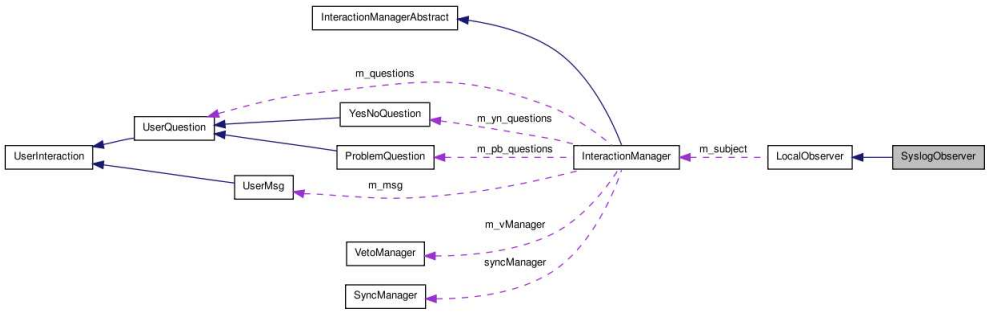


Figure E.5: Interraction diagram of the Syslog Observer

Although there is no point in detailing the Fig. E.6 which is the interaction diagram of the AnalyzerEngine when performing an analysis, it is interesting to see that every single component of the Core software is involved in the process. This illustrates the overall complexity of the logic behind this single method.

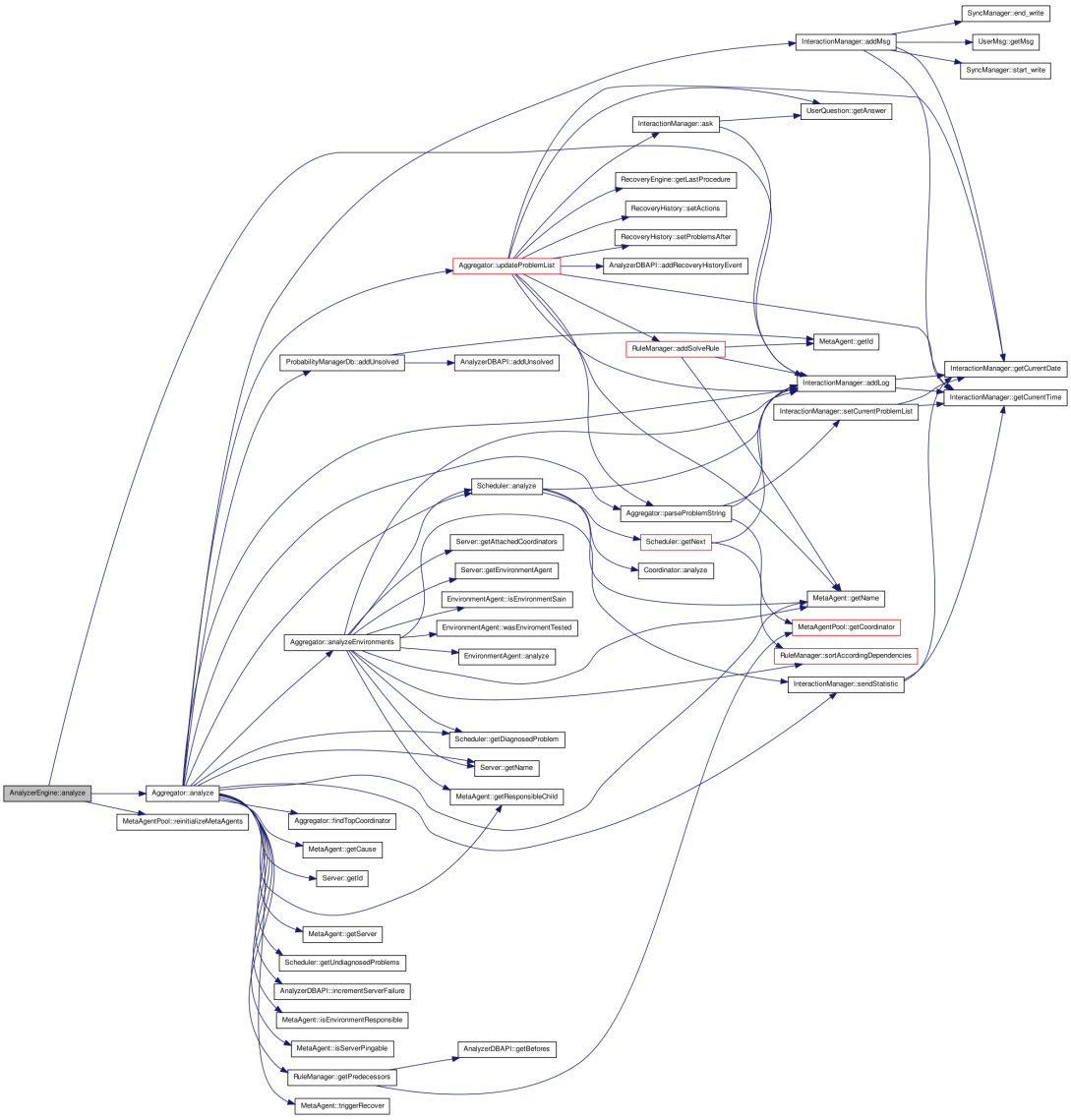


Figure E.6: Interraction diagram of the AnalyzerEngine