



INSTITUT **SUPERIEUR**
D'INFORMATIQUE
DE **MODELISATION**
ET DE LEURS **APPLICATIONS**

COMPLEXE DES CEZEAUX
BP125 – 63173 AUBIERE CEDEX



CERN
European Organisation for
Nuclear Research
CH-1211 GENEVE 23
Suisse

Rapport de stage de 2ème année

Informatique des Systèmes d'Information et de Production
et Aide à la Décision

Supervision de l'écriture de données de l'expérience LHCb

Présenté par : **Chourouk FANANE**
Responsable CERN : **Radu STOICA**
Responsable ISIMA : **Emmanuel MESNARD**

Du 14 Avril
au 12 Septembre 2008





INSTITUT **SUPERIEUR**
D'INFORMATIQUE
DE **MODELISATION**
ET DE LEURS **APPLICATIONS**

COMPLEXE DES CEZEAUX
BP125 – 63173 AUBIERE CEDEX



CERN
European Organisation for
Nuclear Research
CH-1211 GENEVE 23
Suisse

Rapport de stage de 2ème année

Informatique des Systèmes d'Information et de Production
et Aide à la Décision

Supervision de l'écriture de données de l'expérience LHCb

Présenté par : **Chourouk FANANE**
Responsable CERN : **Radu STOICA**
Responsable ISIMA : **Emmanuel MESNARD**

Du 14 Avril
au 12 Septembre 2008

Remerciements

Tout d'abord je tiens à remercier mon tuteur au CERN, Radu STOICA , pour les outils qu'il m'a fournis, pour son écoute et pour avoir patiemment supervisé mon intégration au jour le jour dans un environnement qui m'était complètement étranger.

Je tiens également à remercier Emmanuel MESNARD pour être venu me rendre visite dans les locaux du CERN.

Je remercie enfin Markus FRANCK et Niko NEUFELD pour leur collaboration et leur disponibilité.

Résumé

Situé sur la frontière franco-suisse près de Genève, le **CERN** est aujourd'hui le plus grand laboratoire de recherche en physique fondamentale au monde. Combinant des recherches théoriques et recherches expérimentales, il accueillera en mi-Septembre 2008, le LHC, le plus grand collisionneur de particules, jamais conçu, sur lequel s'articulent plusieurs expériences et en particulier le **LHCb**. C'est justement dans cette phase de conception que prend cadre le travail réalisé d'avril à septembre 2008.

Le but étant de développer un système de **supervision** de l'**écriture de données** venant du système d'acquisition de l'expérience **DAQ**.

L'écriture de données est la dernière étape du système d'acquisition et est une partie très importante du succès de l'expérience. C'est un système distribué entièrement redondant, composé de diverses tâches et de services collectés par une base de données dédiée. Ils servent à manipuler toutes les actions impliquées de la création, l'écriture, la gestion et le transport des fichiers au système de retraitement «offline».

Le projet a été développé en utilisant la librairie **PyDIM**, associée au langage **Python** et le système de communication **DIM**, et le framework **SCADA PVSS**.

Le système réalisé répond maintenant aux exigences et sera prochainement intégré dans le système de contrôle de l'expérience (**ECS**), le système de type SCADA utilisé pour la surveillance et le contrôle de l'ensemble des équipements de l'expérience LHCb.

Mots clés : CERN, LHCb, supervision, écriture de données, PyDIM, DIM, Python, SCADA, PVSS, ECS

Abstract

Located on the French-Swiss border near Geneva, **CERN** is one of the world's biggest scientific laboratories in particles physics, home for both theoretical and experimental research. At CERN the world's most powerful particle collider was build and is now being commissioned, the Large Hadron Collider (LHC).

The LHC, designed as a ring collider, is hosting several big physics experiments, including the Large Hadron Collider beauty experiment (**LHCb**). The LHC, together with the LHCb experiment are expected to go officially into operation in the middle of September 2008. It is precisely in this commissioning phase of the LHCb detector that my internship between April and September 2008 takes place.

The aim of my project is to implement a **monitoring** system for the event **data writing** of physics data coming from the LHCb **DAQ** System.

Event data writing is the last stage of the LHCb DAQ System and is a crucial part for the success of the experiment. It is a fully redundant distributed system composed of various tasks and services gathered around a dedicated database. Their purpose is to handle all actions involved with the file creation, writing, management and transport to the offline reprocessing system.

From a technical point of view, the project relies upon the **Python** language, the **DIM** client-server inter process communication protocol (exposed as a the **PyDIM** library in the Python language) and the **PVSS SCADA** framework.

The monitoring software I have built fulfils the project requirements and is now being integrated with the Experiment Control System (**ECS**) of the experiment, a SCADA system used to monitor and control the whole LHCb detector.

Key-words: CERN, LHCb, monitoring, data writing, DAQ, Python, PyDIM, DIM, PVSS, SCADA, ECS

Glossaire

- **Accélérateur** : Machine qui accélère des faisceaux de particules et les porte à des énergies élevées. On utilise des champs électriques pour accélérer les particules et des champs magnétiques pour les guider et les focaliser.
- **ALICE ou A Large Ion Collider Experiment** : Expérience du LHC créée pour étudier les collisions d'ions lourds.
- **Antimatière** : A toute particule correspond une « antiparticule ». Une antiparticule chargée porte une charge électrique opposée à celle de son homologue de matière. Bien que les antiparticules soient excessivement rares dans l'Univers aujourd'hui, on pense que matière et antimatière ont été créées en quantités égales lors du Big Bang.
- **API ou Application Programming Interface** : Interface pour la programmation des applications, c'est un ensemble de bibliothèques permettant une programmation plus aisée vu que les fonctions deviennent indépendantes du matériel.
- **ATLAS ou A Large ToriDal LHC ApparatuS** : Expérience du LHC visant à étudier les collisions de protons.
- **Big Bang** : Nom donné à l'explosion originelle de l'Univers.
- **CERN** : Au départ l'abréviation CERN signifiait Conseil Européen pour la Recherche Nucléaire, cet acronyme a été conservé bien que le nom officiel du laboratoire soit désormais Organisation Européenne pour la Recherche Nucléaire.
- **Collisinoeur** : Type spécial d'accélérateur dans lequel on accélère deux faisceaux circulant en sens inverses, qui interagissent en des points de collision déterminés. L'énergie de collision dépasse ainsi largement celle que l'on obtient avec un faisceau individuel, dans des collisions avec une cible fixe.
- **CMS ou Compact Muon Solenoid** : Expérience du LHC étudiant les collisions entre protons.
- **DAQ ou Data Acquisition** : Sous système de LHCB ONLINE qui concerne tout le trafic entre les électroniques et les détecteurs du LHCB.
- **DIM ou Distributed Information Management System** : Système de communication pour environnements distribués/mixtes, il fournit une couche transparente de communication d'interprocessus de réseau
- **DNS ou Domain Name System** : Système permettant d'établir une correspondance entre une adresse IP et un nom de domaine dans le but de trouver une information à partir du nom de domaine.
- **DOM ou Document Object Model** : API permettant la manipulation de fichiers XML

- **DP ou Data Point** : Point De Données, des variables de PVSS utilisées pour conserver les données nécessaires.
- **DPE ou Data Point Element** : Élément de Points de Données. Il peut être sous forme d'un nœud (structure) ou d'un type classique (entier, décimale, caractère etc.) .
- **ECS ou Experiment Control System** : Système de contrôle de l'expérience LHCb.
- **Ferme** : Lieu de stockage de serveurs.
- **Framework** : (litt. cadre de travail) en génie logiciel il s'agit d'un ensemble de composants mis à disposition pour faciliter le travail de développement.
- **FMC ou Farm Monitoring and Control system** : Système de contrôle et surveillance de la ferme de l'expérience LHCb.
- **Hadron** : Particule faite de constituants d'interaction forte (quarks/gluons).
- **JCOP Frame Work** : Joint Control Project est une structure logicielle développée au CERN qui contient tous les composants disponibles permettant de les intégrer dans d'autres projets.
- **LHC ou Large Hadron Collider** : (Grand Collisionneur de Hadrons), machine développée au CERN pour effectuer des collisions de protons à très haute énergie. Sa mise en fonctionnement devrait s'effectuer en 2008.
- **LHCb ou LHC beauty** : Détecteur de beauté du grand collisionneur d'hadrons.
- **Méson B** : Particule composée d'un quark b et de son antiquark $\bar{b}$.
- **Orienté Objet** : Langage de programmation reposant sur la manipulation de briques logicielles appelées objet.
- **PVSS ou Process Visualisation and control system** : Système de contrôle et de visualisation de processus, produit industriel, de type SCADA .
- **Quart** : Particules élémentaires de la matière.
- **SCADA ou Supervisory Control And Data Acquisition** : Systèmes logiciels commerciaux utilisés intensivement dans l'industrie pour la surveillance et le contrôle des processus industriels.
- **TCP/IP ou Transmission Control Protocol/Internet Protocol** : Protocole utilisé sur le réseau Internet pour transmettre des données entre deux machines. TCP, protocole de transport, prend à sa charge l'ouverture et le contrôle de la liaison entre deux ordinateurs. IP, protocole d'adressage, assure le routage des paquets de données .
- **Trigger** : Dans la chaîne d'acquisition de données, les triggers permettent de sélectionner les événements à retenir, dans le but de réduire la quantité de données à traiter et stocker.

- **UML ou Unified Modeling Language** : Langage de modélisation pour la programmation orientée objet.
- **XML ou eXtensible Markup Language** : Langage informatique de balisage générique.

Table des matières

Introduction.....	1
1. Présentation générale.....	2
1.1 CERN.....	2
1.2 L'expérience LHCb.....	3
1.2.1 Le LHC.....	3
1.2.2 Description de l'expérience LHCb	4
1.2.3 Les différents systèmes mis en jeu dans l'expérience	6
2. Analyse du problème.....	8
2.1 Étude de besoin.....	8
2.2 Prise en main des outils logiciels.....	9
2.2.1 Python	9
2.2.2 DIM	9
2.2.3 PVSS	11
2.2.4 Communication DIM/PVSS.....	15
2.3 Prise en main de l'existant.....	16
2.3.1 la librairie FwDIM.....	16
2.3.2 La librairie PyDIM.....	16
3 Réalisation de la solution.....	17
3.1 Définition de l'environnement du travail.....	17
3.2 Démarche suivie pour le développement de l'application.....	18
3.2.1 Création du fichier XML et extraction de données.....	18
3.2.2 Développement d'un système de contrôle et de supervision.....	21
3.2.3 Création du panneau de supervision en PVSS.....	26
4 Résultats et discussions.....	29
4.1 Résultats.....	29
4.2 Enrichissement personnel.....	29
Conclusion.....	31
Références bibliographiques	32

Table des figures

Figure 1: Photographie aérienne du CERN	3
Figure 2: Photo du collisionneur des hadrons.....	4
Figure 3: Schéma du détecteur LHCb.....	5
Figure 4: Schéma du système de contrôle de l'expérience.....	7
Figure 5: Schéma général de l'application.....	8
Figure 6: Fonctionnement de DIM.....	10
Figure 7: Gestionnaire de tâches de PVSS.....	11
Figure 8: Exemple de deux DP de type HvChannel.....	13
Figure 9: Éditeur graphique de PVSS.....	14
Figure 10: L'environnement de l'application de contrôle.....	17
Figure 11: Diagramme de classe représentant les ordinateurs de contrôle.....	18
Figure 12: Extrait de l'initial fichier XML créé.....	19
Figure 13: Fonctionnement de L'API DOM.....	20
Figure 14: Diagramme de classe de la structure du développement.....	22
Figure 15: Exemple des services publiés par DIM.....	23
Figure 16: Résultat du premier contrôle publié par DIM.....	24
Figure 17: Résultat du 2ème contrôle publié par DIM.....	26
Figure 18: Connexion du service au DataPoint.....	27
Figure 19: Extrait du script PVSS de la création du pannel.....	28
Figure 20: Panneau de supervision des taskMonitorings.....	29
Figure 21: Résultat final du contrôle.....	30

Introduction

La physique des particules est une science qui cherche à comprendre et à décrire la structure de la matière. Pour pouvoir étudier et confronter les modèles théoriques aux résultats expérimentaux, des détecteurs sont construits autour des points d'interaction des accélérateurs de particules. Aujourd'hui, quatre nouveaux détecteurs sont installés sur le nouvel accélérateur LHC (Large Hadron Collider) du CERN (Organisation Européenne de la Recherche Nucléaire).

Le détecteur LHCb est l'un de ces quatre détecteurs. Il a pour objectif de comprendre la différence de comportement entre une particule et son anti-particule. Il est constitué de plusieurs sous-détecteurs destinés à l'identification des particules produites lors des collisions.

Dans le cadre l'expérience LHCb, des événements de physique sont sélectionnés par une large ferme de calcul. Ces événements sont ensuite enregistrés dans des fichiers puis sauvegardés dans un système de stockage de très haute puissance et de haute disponibilité.

Un fichier d'environ 2 Go sera créé toutes les 30 secondes. Ces fichiers doivent être copiés à un système de bande utilisé pour le stockage à long terme. Ils sont aussi pré-traités et utilisés par plusieurs autres processus en temps réel, qui vérifient la qualité des données.

Mon travail portait sur le développement d'un système de contrôle et surveillance de ces processus, l'implémentation des panneaux de supervision visuel et l'intégration de ce système dans le Système de Contrôle de l'Expérience (ECS) de LHCb.

Ce rapport décrit d'abord le contexte scientifique et technique du travail effectué. Ensuite l'ensemble des outils y est présenté afin de donner un aperçu des composants impliqués. Enfin les travaux effectués et leurs résultats sont décrits et discutés dans les deux dernières parties.

1. Présentation générale

1.1 CERN

Fondé en 1954 par une convention entre 12 états européens, le CERN , l'Organisation Européenne pour la Recherche Nucléaire, est le plus grand centre mondial de recherche en physique des particules. Il est situé sur la frontière franco-suisse, près de Genève. C'est un laboratoire où les physiciens développent des expériences pour explorer les constituants élémentaires de la matière et étudier les forces qui assurent leur cohésion. Il fournit aux physiciens des outils performants : des accélérateurs, pour porter les particules jusqu'à une vitesse proche de celle de la lumière, et des détecteurs pour observer les particules produites lors des collisions.

Plus de 80 nationalités sont présentes au sein du CERN, 2500 personnes sont mobilisées en alliance physiciens, ingénieurs, techniciens, ouvriers administrateurs ou secrétaires, qui sont entre autres chargés de concevoir et de construire les appareils sophistiqués mis en œuvre dans chaque expérience : le collisionneur LHC et des détecteurs.

Quatre grandes expériences LHCb, Atlas, Alice et CMS se préparent simultanément, chacune étudie un aspect précis des particules lors de l'accélération et de la collision. Elles ont lieu en France et en Suisse sur un périphérique circulaire d'un diamètre de 27 Km.

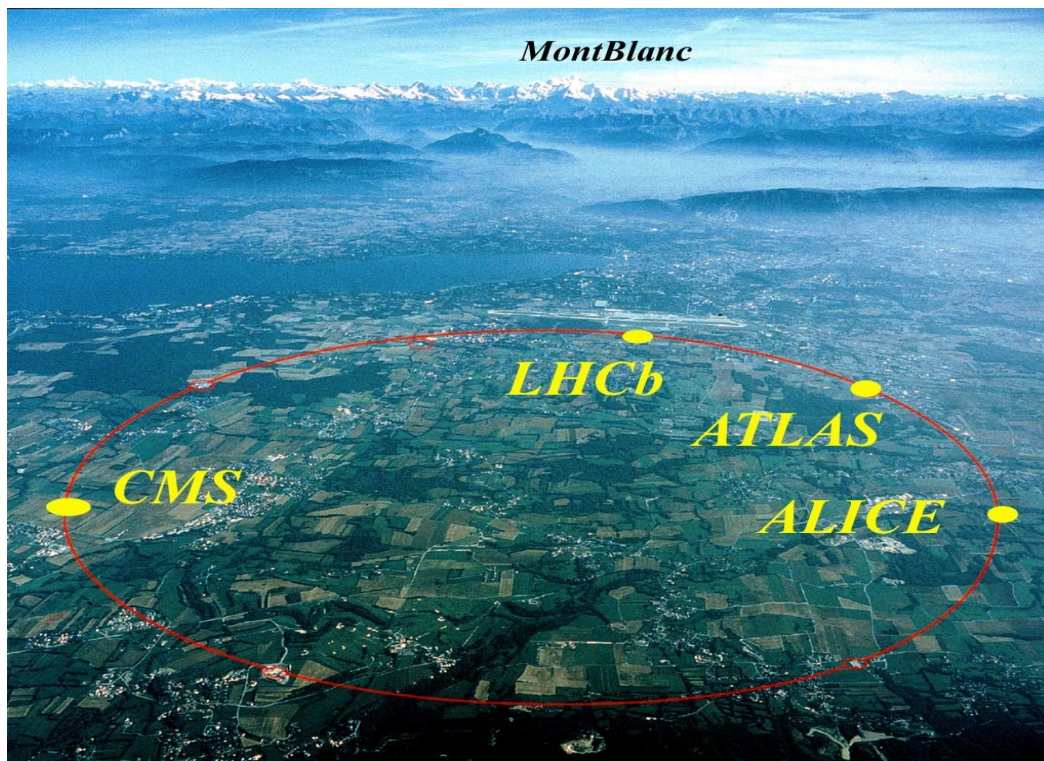


Figure 1: Photographie aérienne du CERN

1.2 L'expérience LHCb

1.2.1 Le LHC

Le Grand collisionneur de hadrons (LHC) est un énorme instrument scientifique situé près de Genève, sur la frontière franco-suisse, est enterré à environ 100 m de profondeur. C'est un accélérateur de particules, avec lequel les physiciens vont étudier les plus petites particules connues : les composants fondamentaux de la matière. Le LHC révolutionne notre compréhension du monde, de l'infiniment petit, à l'intérieur des atomes, à l'infiniment grand de l'Univers.

Il est destiné à faire entrer en collision des protons à une fréquence de 40 MHz. Des faisceaux de noyaux de plomb seront également accélérés, entrant en collision avec une énergie de 1150 TeV.

Tout d'abord, un accélérateur linéaire est utilisé afin d'atteindre une énergie de 50 MeV. Ensuite, deux accélérateurs circulaires sont utilisés pour augmenter l'énergie des particules jusqu'à 26 GeV avant leur entrée dans le Super Proton Synchrotron et dans le LHC. L'énergie d'entrée dans le LHC des particules est alors de 450 GeV et elle est portée à 7 TeV.

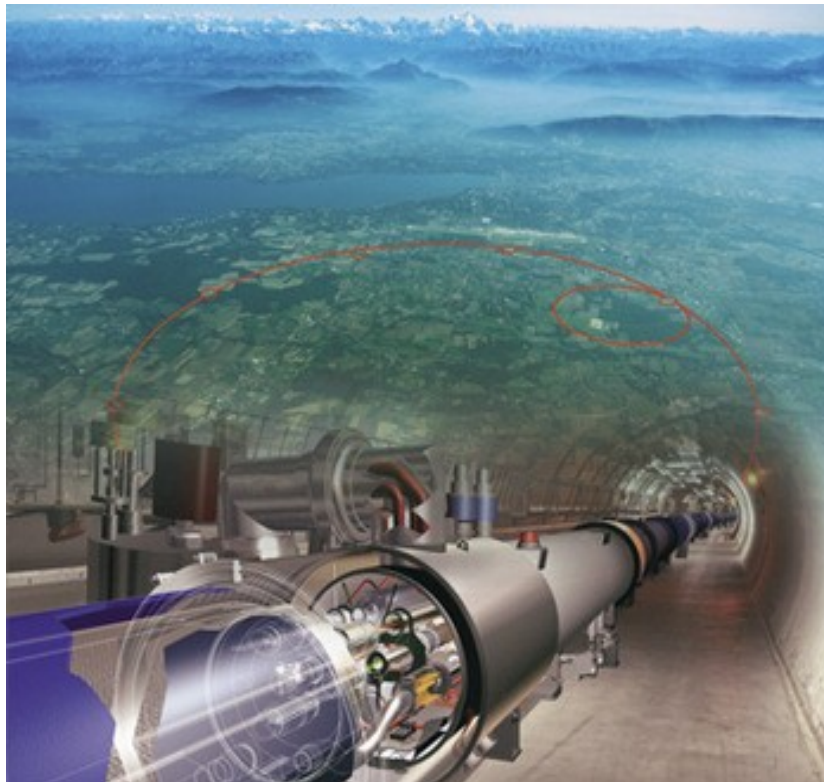


Figure 2: Photo du collisionneur des hadrons

Sur cet accélérateur, 3 expériences de physique des particules seront menées, ATLAS, CMS et LHCb, et une expérience de physique nucléaire hadronique ALICE destinée à la production et à l'exploration des propriétés du plasma de quarks et de gluons. Ce plasma est un état de la matière qui a existé à une fraction de seconde après le big-bang.

1.2.2 Description de l'expérience LHCb

L'expérience LHCb (Large Hardon Collider Beauty experiement) a pour but d'explorer les différences entre matière et antimatière en étudiant un type de particule appelée « quark beauté » ou « quark b ». Elle utilise une série de sous-détecteurs alignés le long du faisceau afin de traquer principalement les particules à petits angles. Le premier sous-détecteur est installé près du point de collision, les autres se suivent sur une longueur de 20 m.

- **Objectif**

Après le Big Bang, toute la matière aurait dû être annihilée par l'antimatière, mais heureusement pour nous, la nature a favorisé la matière. Une différence de comportement entre la matière et l'antimatière a été déjà observée, mais elle est loin d'expliquer l'excès de matière dans l'univers primordial. L'expérience LHCb est conçue

pour élucider ce mystère.

Le LHC va accélérer et faire entrer en collision des particules avec les énergies les plus élevées jamais atteintes en laboratoires. Le détecteur LHCb va enregistrer ces collisions recréant les conditions de l'Univers lorsqu'il n'était âgé que d'une centième de milliardième de seconde.

Les paires de quarks et d'antiquarks produites dans les collisions vont fuser selon les trajectoires proches de la ligne de faisceau. C'est pourquoi l'expérience LHCb a été conçue comme une série de sous-détecteurs placés les uns derrière les autres, au plus près de l'accélérateur sur une longueur de plus de 20 mètres.

L'objectif du LHCb est donc de détecter une asymétrie plus importante qui aiderait à comprendre pourquoi la nature préfère la matière à l'antimatière.

- **Le détecteur LHCb**

Bien que très grand et très lourd (21 mètres de long, 13 mètres de large et 10 mètres de haut, 5600 tonnes, à 100 mètres sous terre dans la région Ferney-Voltaire, France), le détecteur LHCb est un instrument à haute précision, basé sur les technologies les plus avancées. Sa taille tient au fait qu'il consiste en une succession de différents types de sous-détecteurs, chacun spécialisé dans la mesure d'une caractéristique spécifique de particules issues de la collision. Dans son ensemble, le détecteur fournit des informations sur l'identité, la trajectoire, la quantité de mouvement et l'énergie des particules produites dans les collisions. Chaque sous détecteur est également très grand afin d'effectuer des mesures précises des particules produites, extrêmement rapides et énergétiques.

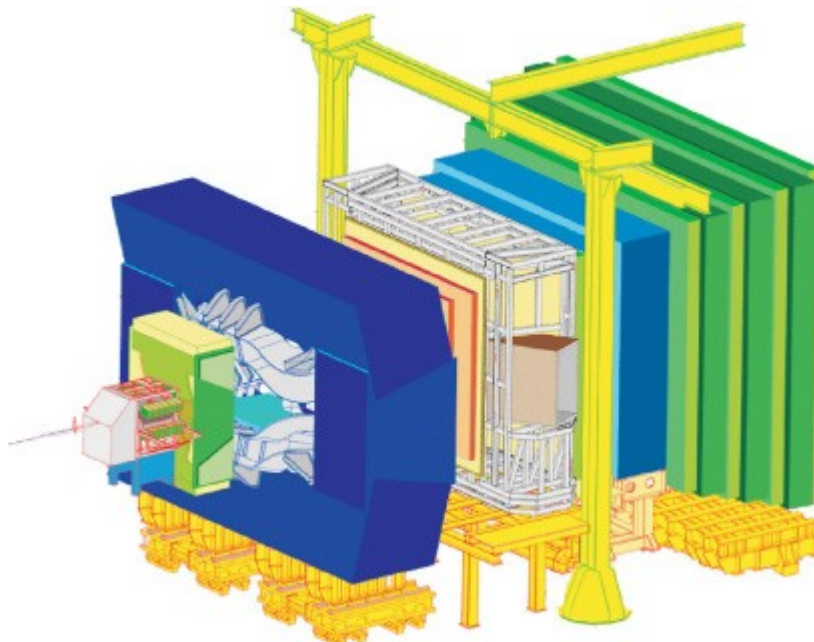


Figure 3: Schéma du détecteur LHCb

1.2.3 Les différents systèmes mis en jeu dans l'expérience

Plusieurs systèmes sont mis en jeu dans l'expérience LHCb, afin de procéder à l'acquisition des données issues du détecteur, comme au contrôle du bon déroulement de l'expérience.

- **Trigger**

Les collisions ont lieu à une fréquence de 40 MHz (une collision toutes les 25 ns), ce qui représente une grande quantité de données à traiter. Le «Trigger System» qui est traduit par «système de déclenchement» ou «prise de décision», a pour but de réduire cette fréquence en ne sélectionnant que les événements importants. Il repose sur trois niveaux différents de « filtrage » des données, dont le rôle est de prendre des décisions rapides quant à l'importance à apporter à chacune des collisions, dans l'optique de les garder ou de les rejeter.

- **DAQ**

Le système DAQ (Data AcQuisition) [TDR] s'étend de l'électronique qui interface les sous-détecteurs de l'expérience LHCb jusqu'au système de stockage des données potentiellement intéressantes. Son but est de filtrer l'énorme rendement du détecteur (40TB/s données en entrée, 75MB/s En sortie).

Tout au long de ce système, les données sont sélectionnées par différents mécanismes pour ne conserver que ce qui est potentiellement pertinent et intéressant pour l'expérience.

Une chaîne complexe de traitement est mise en œuvre allant de la détection des particules, de la mesure de leurs paramètres caractéristiques jusqu'à une analyse statistique complète. Les principales étapes de la chaîne de traitement sont de :

- détecter les particules produites lors de la collision et de produire des données à partir d'un dispositif électronique complexe
- réduire et stocker les informations produites sur un support informatique
- analyser ensuite ces données par une ferme de calcul
- enregistrer ces données dans des disques de stockage

Le système d'acquisition de données se compose de :

- **Front-End electronics** : une électronique frontale qui s'interface avec le dispositif de sélection des collisions.
- **Readout Network** : un réseau de lecture qui permet un filtrage matériel de données.
- **Processing/Filtering** : une ferme de déclencheurs qui traitent les données
- **Storage** : un système de stockage qui permet de sauvegarder les fichiers où les données sont enregistrées.

- **ECS**

Le système de contrôle de l'expérience LHCb, ECS (Experiment Control System) [ECS] est en charge de paramétrer, configurer , et surveiller l'ensemble des

équipements de l'expérience :

- Système d'acquisition des données et de déclenchement
- Système de contrôle du détecteur DCS (Detector Control System) : gaz, alimentation haute et basse tension, température, etc
- Infrastructure expérimentale : l'aimant, la ventilation, la distribution d'électricité, dispositif de sécurité, etc
- Interaction avec l'accélérateur LHC, le système de sécurité du CERN, les services techniques du CERN, etc

Les relations entre l'ECS et les différents composants de l'expérience sont illustrées en figure ci-dessous. Cette figure montre que l'ECS fournit une unique interface entre les utilisateurs et les différents équipements de l'expérience.

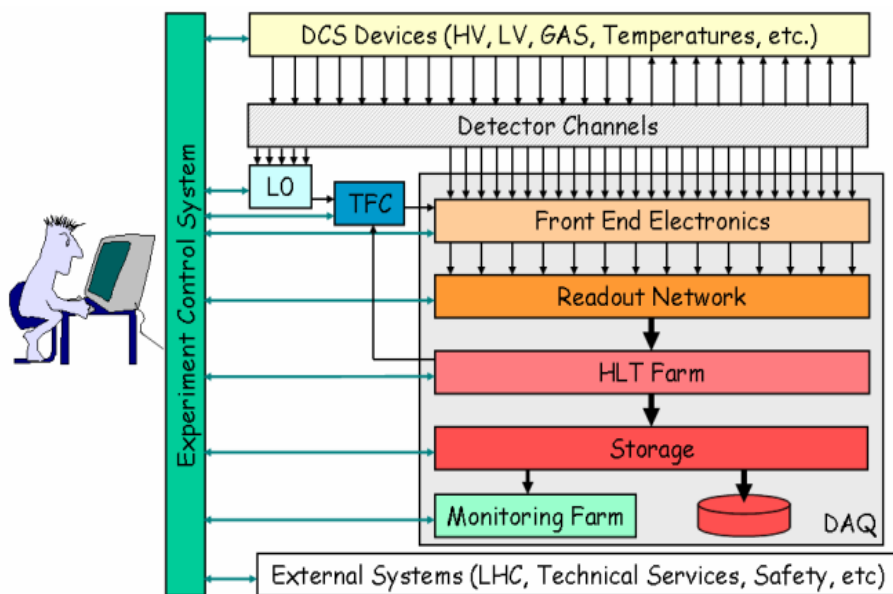


Figure 4: Schéma du système de contrôle de l'expérience

La structure générale du contrôle est basée sur un système de type SCADA (Supervisory Control and Data Acquisition) appelé PVSSII, qui permet la gestion à grande échelle de chaîne de mesure et de contrôle. Il est utilisé pour interfacer les systèmes électroniques avec le système de contrôle et effectuer leur surveillance, leur contrôle, leur configuration et l'initialisation de leurs paramètres.

Pour le contrôle, une structure logicielle commune à toutes les expériences est développée, le JCOP FrameWork (Joint COntrol Project) dans laquelle les membres du LHCb ont largement contribué. Cette structure utilise aussi PVSS II, qui a largement facilité le développement de dispositifs de contrôle. Cette structure commune devrait permettre le développement d'une interface cohérente et homogène entre toutes les expériences, ce qui simplifiera par la suite la tâche des équipes dans la mise en route du système, la détection des problèmes et leur résolution.

2. Analyse du problème

2.1 Étude de besoin

Comme nous l'avons vu précédemment, une fois issus du détecteur LHCb, des événements vont être collectés par le système d'acquisition de données. Il va ensuite les filtrer puis les stocker temporairement dans des mémoires analogiques ou numériques. Les données seront ensuite sélectionnées par les PCs de la ferme, enregistrées dans des fichiers et définitivement enregistrées en cas d'acceptation de la collision dans des disques de stockage.

La figure ci dessous illustre bien ce parcours et contrôle de données.

Les tâches qui traitent la manipulation de ces fichiers peuvent être exécutées dans des différentes machines qui communiquent entre elles et au ECS à travers DIM.

Parmi les problèmes d'ordre informatique qu'on peut rencontrer, c'est le contrôle de la manipulation de ces fichiers. De ce fait, mon travail a consisté à contrôler toutes ces tâches depuis le système de contrôle de l'expérience ECS.

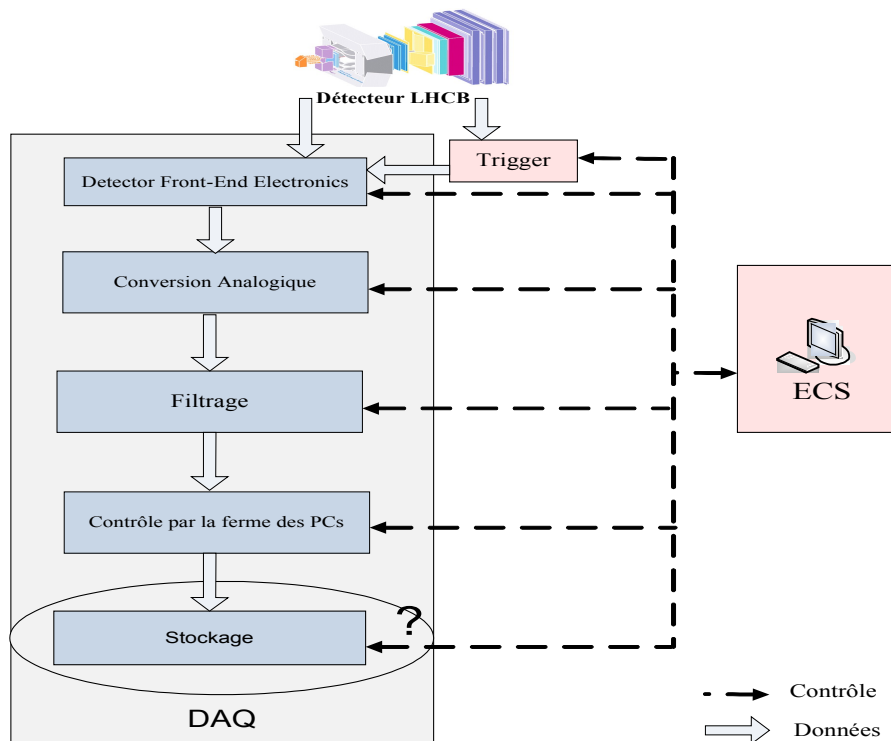


Figure 5: Schéma général de l'application

2.2 Prise en main des outils logiciels

2.2.1 Python

Python est un langage de programmation facile à utiliser et puissant. Il offre des structures de données puissantes de haut niveau et une approche simple mais réelle de la programmation orientée-objet. La syntaxe élégante de python est le typage dynamique, ajoutée à sa nature interprétée, en font un langage idéal pour écrire des scripts et pour le développement rapide d'applications dans de nombreux domaines et sur la plupart des plates-formes.



Python fait partie, avec le C et le C++, des langages communément utilisés au CERN. Mais il a été préféré pour sa simplicité qui rend le processus de développement beaucoup plus rapide. En effet, en combinant une syntaxe très simple à un ensemble de structures de données évoluées (listes, dictionnaires, ...), Python permet de produire des programmes à la fois très compacts et très lisibles. A fonctionnalités égales, un programme Python abondamment commenté est souvent de 3 à 5 fois plus court qu'un programme C ou C++ équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.

2.2.2 DIM

DIM (Distributed Information management) [DIM] est un système de communication pour environnements distribués/mixtes, il fournit une couche transparente de communication d'interprocessus de réseau. C'est un système, développé au CERN, permettant de fournir une couche de communication inter-processus transparente, à travers un réseau basé sur le protocole TCP/IP.

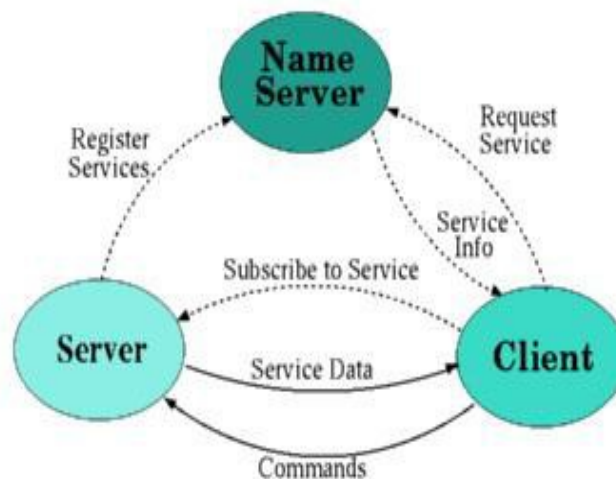


Figure 6: Fonctionnement de DIM

● Fonctionnement de DIM

Le concept de base dans DIM est le concept du « service ». Les serveurs fournissent des services aux clients. Un service est normalement un ensemble de données (de n'importe quel type ou taille) et il est identifié par un nom - « des services appelés ». L'espace nommé pour des services est libre.

– chaque serveur se fait connaître auprès d'un DNS, DIM Name Service auprès duquel il inscrit la liste des services et des commandes qu'il fournit. En d'autres termes, le DNS enregistre tout ce que le serveur est capable et se propose de faire.

– chaque client, peut souscrire à un service ou à une commande que fournit n'importe quel serveur, à condition qu'il soit connecté au même DNS que lui. Pour cela, le client demande au DNS quelle est l'adresse du serveur qui fournit un certain service ou une certaine commande.

Celui-ci lui renvoie l'adresse, puis le client se connecte directement au serveur en question. Si le serveur est momentanément indisponible, le DNS ne répond rien, mais garde en attente la demande du client, et dès que le serveur approprié se reconnecte, le DNS prévient le client demandeur.

Une fois connectés l'un avec l'autre, le serveur et le client peuvent s'échanger des données dans un sens comme dans l'autre sans aucun intermédiaire, le DNS ne servant qu'à la mise en relation des deux entités.

● Utilités de DIM

Ce système est particulièrement efficace dans le cadre d'un système réparti entre plusieurs machines différentes, car si un serveur est amené à changer physiquement de place, il n'est pas nécessaire d'avertir un à un les clients susceptibles d'entrer en contact avec lui de son changement d'adresse. En effet, à sa reconnexion, le DNS va noter que le serveur aura une nouvelle adresse, qui sera alors celle qu'il transmettra.

DIM fonctionne dans des environnements de travail aussi bien sous Windows que sous Linux.

DIM offre une librairie de fonctions qui peuvent être intégrées dans un code C++ ou Java, cette librairie est principalement composée de deux types de classes :

- Classes du serveur : à utiliser par les serveurs de DIM, elles implémentent des méthodes statiques relatives au serveurs et permettent de créer des serveurs, des services et des commandes ainsi qu'effectuer la mise à jours des services à partir d'un objet serveur.
- Classes du client : implémentent des méthodes relatives aux clients, contient plusieurs méthodes statiques, la principale est l'envoi d'une commande par un client à un serveur ou la demande d'informations à un serveur et leur mise à jour.

2.2.3 PVSS

Le système de contrôle du LHCb repose sur PVSS [PVSS] qui est un logiciel de type SCADA (Supervisory Control And Data Acquisition, Superviseur de contrôle et d'acquisition de données).



L'application PVSS est très hiérarchisée et est composée de différents processus s'exécutant en parallèle. Comme on peut le constater d'après la figure ci-dessous, cette application se compose de plusieurs gestionnaires de tâche :

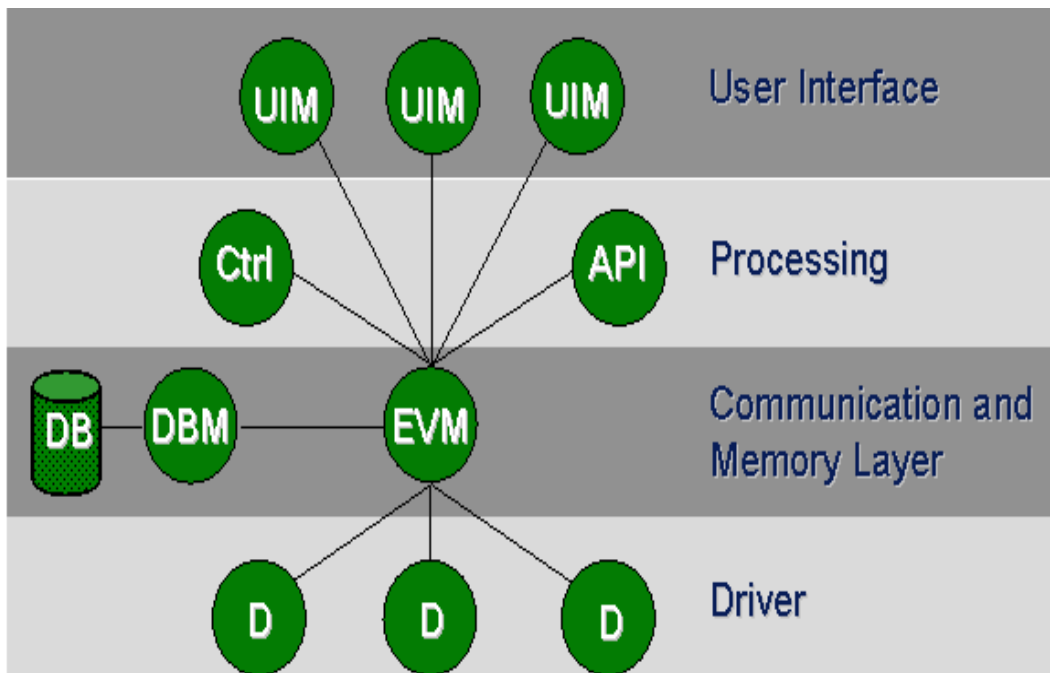


Figure 7: Gestionnaire de tâches de PVSS

- un gestionnaire d'événement EVM (Event Manager) responsable de la gestion de toutes les communications. Il reçoit les données provenant des pilotes (D Driver) et stocke les données dans la base de données ;
- un gestionnaire de la base de données (DBM DataBaseManager) qui fournit une interface avec la base de données ;
- un gestionnaire d'interface graphique de l'utilisateur (UIM User Interface Manager) permettant d'obtenir les données de différents systèmes ou de stocker les données dans la base de données à envoyer au système ;
- un gestionnaire de contrôle (Ctrl Manager) permettant d'effectuer le traitement des données, ce langage est une extension du langage C ;
- un gestionnaire API (Application Programming Interface) permettant à chaque utilisateur d'écrire son propre programme C++ pour avoir accès aux données de la base de données ;
- des pilotes (D Driver) permettant l'interface des systèmes à contrôler.

PVSS fonctionne soit sous un système Windows soit sous un système Linux. L'application PVSS peut également être distribuée à travers différentes machines. Cette distribution du système est construite en ajoutant un gestionnaire de distribution pour chaque système. De cette manière un certain ensemble de systèmes peuvent alors être connectés.

PVSS permet de modéliser les composants en utilisant le concept de DataPoint. Ce concept est principalement basé sur la notion d'objet. La modélisation s'effectue en deux étapes.

D'abord, un DPE DataPoint type doit être défini par l'utilisateur. Il décrit la structure de données associée au composant à modéliser. Par analogie aux langages orientés objet, on pourrait définir un DataPoint type comme étant une classe. Un DataPoint type peut contenir des entiers, des chaînes de caractères ou bien même des pointeurs sur d'autres structures.

Ensuite, l'utilisateur peut définir autant d'instances de DataPoint type qu'il souhaite : ce sont les DataPoint Element. Ils représentent un objet particulier et contiennent les données propres à cet objet.

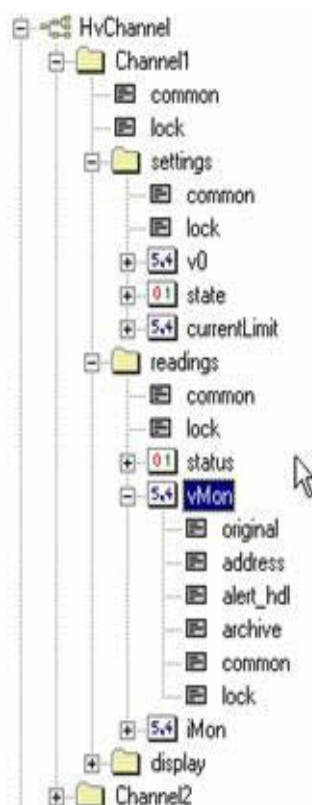


Figure 8: Exemple de deux DP de type HvChannel

Ainsi, il est possible de modéliser un système sous forme d'arbre de données. Une fois la structure créée, toute une gamme d'options est à la portée de l'utilisateur : ce dernier peut mettre en place des contraintes sur certaines valeurs, en particulier des alarmes, ou effectuer des archivages.

En utilisant l'éditeur graphique (ci-dessus), l'utilisateur dans un premier temps définit la partie statique de son panneau en dimensionnant son interface puis en déposant les boutons, les tableaux, les champs de caractères à l'endroit désiré. Ensuite il fait correspondre aux boutons une action pour un évènement donné. Par exemple, l'utilisateur envoie une donnée lorsqu'il clique sur un bouton. Les actions sont programmées en utilisant des scripts où le code est écrit en langage PVSS.

Les scripts PVSS permettent une communication entre l'utilisateur et la base de données. La syntaxe du langage PVSS est très proche de celle du langage C. Une librairie de fonctions assez large permet de manipuler des DataPoint Element, des graphiques ou même encore des fichiers.

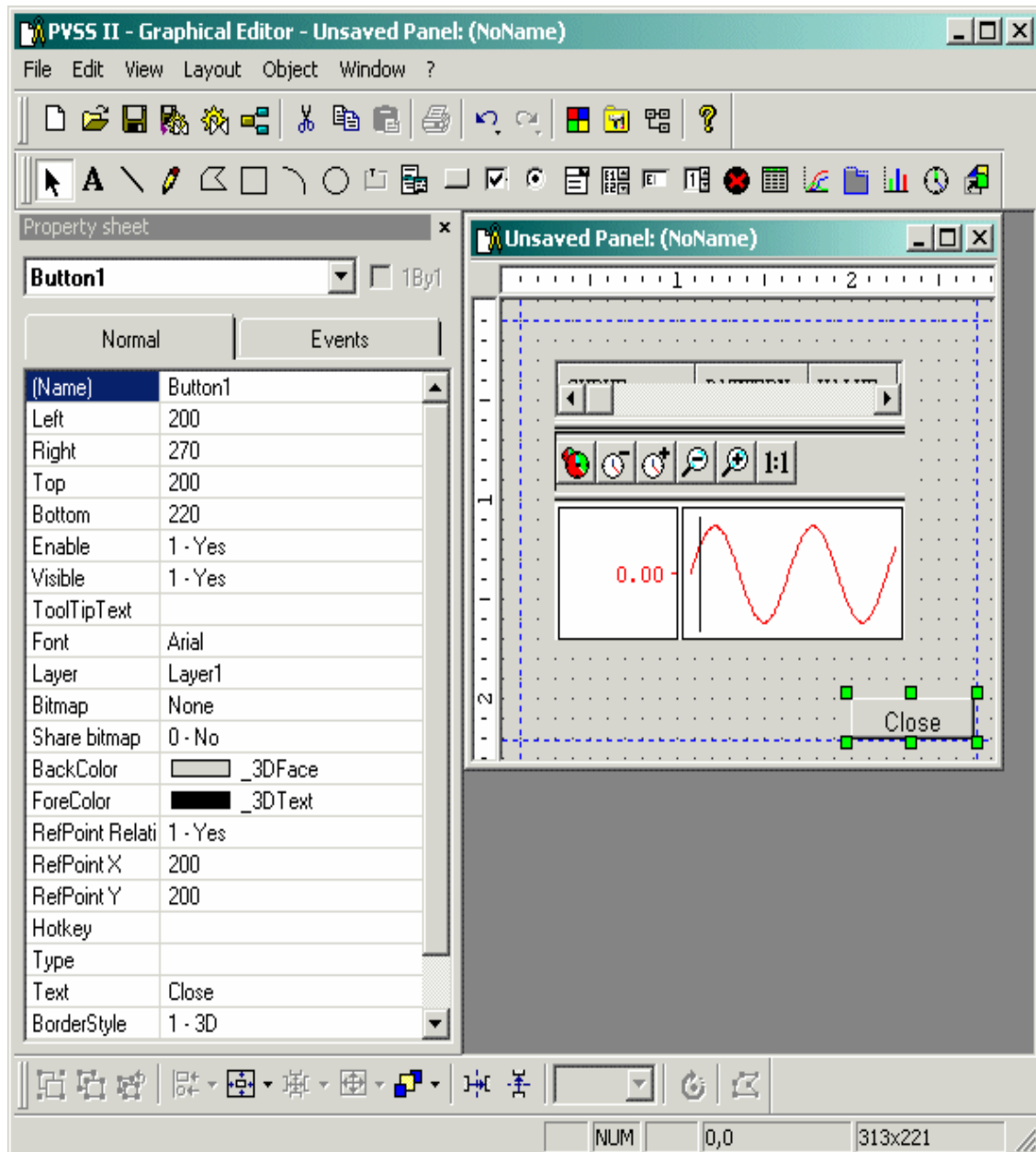


Figure 9: Éditeur graphique de PVSS

Trois fonctions en terme d'accès aux DataPoint Element sont importantes :

- **dpGet** (« nom du DataPoint Element », « variable »)
Cette fonction récupère la valeur du DataPoint Element présente dans la base de données et la place dans la variable indiquée en paramètre.
- **dpSet** (« nom du DataPoint Element », « valeur »)
Cette fonction écrit la valeur dans le DataPoint Element présent dans la base de données.
- **dpConnect** (« nom de la fonction », « nom du DataPoint Element »)

dpConnect permet d'appeler une fonction à chaque fois que le contenu du DataPoint Element passé en paramètre change. Ainsi, si par exemple on souhaite afficher le contenu d'un DataPoint Element en continu, il suffit de passer en paramètre le nom de la fonction d'affichage que l'on aura défini précédemment, accompagné du DataPoint.

L'un des points recherché par les utilisateurs du CERN consiste à pouvoir contrôler toute sorte de composants électroniques à partir d'un ordinateur distant quelconque. Ainsi, l'interface entre PVSS et le matériel n'est pas directe. Le protocole de communication DIM effectue la jonction entre les deux.

2.2.4 Communication DIM/PVSS

Le rattachement du système de communication DIM au système de contrôle PVSS s'effectue par l'intermédiaire d'un API Manager de PVSS, qui à partir de programmes extérieurs écrits en C++, permet d'accéder à la base de donnée interne de PVSS et de la modifier [DIM/PVSS].

La communication entre les processus de DIM se fait via des DataPoints. PVSS représente le client qui envoie des commandes. Chaque commande est reliée à un DataPoint.

Le changement de l'une des DataPoints est pris en compte par DIM qui fait automatiquement sa mise à jour. Une fois qu'un DataPoint a changé de valeur, DIM exécute un envoi de la nouvelle commande vers le serveur qui peut tourner sur une machine distante. La connexion est établie grâce à la variable d'environnement DIM_DNS_NODE, cette variable est fixée par l'utilisateur qui doit fournir au client et au serveur le nom de la machine sur laquelle tourne DIM, la communication est dès lors établie. Il faut noter que sur une machine on ne peut avoir qu'un seul DIM manager. Au niveau de la configuration de DIM il faut aussi affecter chaque service à un DataPoint. Dès que le serveur reçoit les commandes de DIM il les traite et renvoie les résultats de ce traitement à DIM qui les publie dans le DataPoints correspondant à chaque service. PVSS prend ensuite ces services pour effectuer un traitement.

Il existe des fonctions de DIM qui permettent d'exécuter des scripts dès le changement des valeurs de certaines DataPoints. DIM offre un moyen de diagnostic sur une interface graphique indépendante de PVSS, nous parlons ici de DID (Distributed Information Display) c'est un outil qui permet de visualiser la liste des serveurs mis en marche ainsi que la liste des services disponibles. Il donne aussi à l'utilisateur la possibilité de sélectionner un serveur particulier et d'accéder à sa liste de services ou de commandes.

DID est aussi un moyen pour tester le fonctionnement du serveur seul en permettant d'envoyer des commandes indépendamment de PVSS via son interface graphique et de lire le contenu du service où DIM publie les résultats.

2.3 Prise en main de l'existant

2.3.1 la librairie FwDIM

Pour recevoir où envoyer des données via des services et commandes de DIM, PVSS utilise les points de données (DP). Les éléments de ces Points de Données (DPE) doivent être de même type que les valeurs envoyées par le service.. Ces DP ont une forme spécifique et elles servent comme des variables « globales » du système PVSS. Pour s'inscrire à un ou plusieurs services ou commandes, on peut utiliser l'interface graphique qui est un Panneau du FwDIM ou utiliser la librairie de DIM et de créer un nouveau panneau destiné à cette opération.

FwDIM est une librairie de fonctions permettant d'ajouter ou supprimer des services ou des commandes DIM.

Parmi ses principales fonctions:

- fwDim_subscribeService() : pour s'inscrire à un nouveau service
- fwDim_findServices() : pour trouver les services existants dans un DNS donné.
- fwDim_publishService() : pour publier un nouveau service

2.3.2 La librairie PyDIM

C'est une librairie basée sur les classes de DIM et une partie de l'API C++, elle est implémentée en utilisant l'API python. PyDIM permet d'appeler toutes les fonctions de DIM depuis Python en utilisant des classes de C++. Ça accomplit aussi une conversion automatique de C/C++ aux callbacks de Python.

La structure des fonctions de PyDIM a la même sémantique que celle des composants de DIM.

Cette librairie contient deux types de fonctions:

- Méthodes relatives au serveur commençant par «dis», permettant de démarrer un serveur «dis_start_serving()», créer un service «dis_add_service()», mettre à jour un service «dis_update_service()» ...
- Méthodes relatives au client commençant par «dic», permettant demander une information au serveur«dic_info_service()»...

3 Réalisation de la solution

3.1 Définition de l'environnement du travail

Ma première tâche a été de définir un environnement de travail avant de pouvoir commencer le développement de l'application.

Afin de faciliter le contrôle de ces tâches, il a été choisi de créer un fichier XML qui représente les machines contrôlant la manipulation des fichiers ainsi que les différents processus fonctionnant dedans.

La figure suivante illustre bien les différents composants mis en jeu pour le développement de l'application.

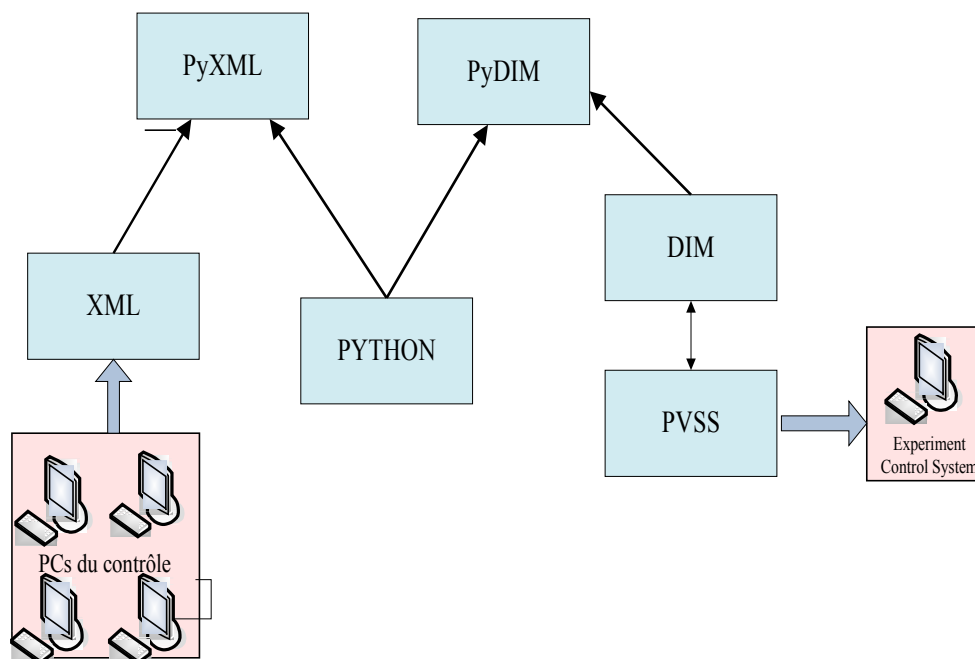


Figure 10: L'environnement de l'application de contrôle

En se basant sur une description XML des tâches effectuées dans chaque ordinateur, j'ai tout d'abord développé une application qui extrait les données du fichier XML et les transforme en objets en utilisant la bibliothèque PyXML.

Ensuite j'ai développé une application qui surveille l'état de chaque tâche en utilisant la librairie PyDIM.

Enfin, J'ai développé une interface graphique en PVSS qui permet de visualiser ce contrôle et afin de l'intégrer dans le système de contrôle de l'expérience ECS.

Ce développement sera détaillé dans la partie suivante.

3.2 Démarche suivie pour le développement de l'application

Mon travail a été effectué en 3 grandes étapes, chacune utilisant des outils différents ce qui m'a permis de manipuler les deux systèmes d'exploitation ainsi de différents outils logiciels.

3.2.1 Création du fichier XML et extraction de données

- **Architecture de l'application**

Pour bien concevoir la structure du contrôle, ce diagramme de classe a été élaboré, en représentant chaque ordinateur par un nœud.

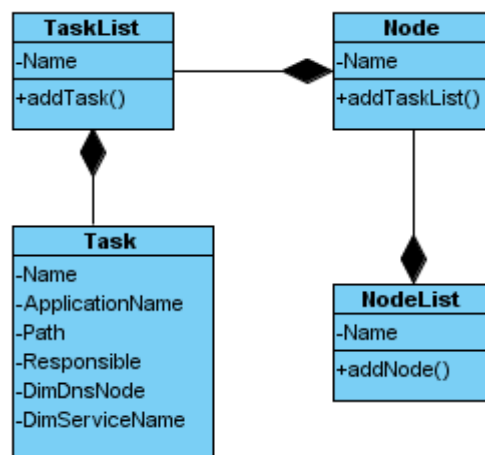


Figure 11: Diagramme de classe représentant les ordinateurs de contrôle

Dans le schéma présenté ci-dessus, on voit que chaque liste des nœuds (NodeList) se compose de plusieurs ordinateurs (Node), et chaque ordinateur a des listes de tâches (TaskList) qui s'exécutent dedans. Et enfin chaque TaskList contient plusieurs tâches (Task) sachant que chaque tâche peut fonctionner dans des nœuds multiples.

- **Création du fichier XML**

Pour contrôler ces tâches, et afin de faciliter une grande portabilité, ces informations ont été stockées dans un fichier XML. Le choix de cette description XML est dû au fait de faciliter la manipulation du contrôle, ainsi l'utilisateur peut déterminer le nombre de tâches à contrôler et peut aisément le modifier.

Dans la cadre de l'application que j'ai développée, voici les données de chaque élément vu précédemment.

- Pour une tâche Task :
 - Name : Nom de la tâche
 - ApplicationName: Nom de l'application qu'elle traite
 - Path : Son chemin
 - Responsable: Nom du responsable de cette tâche
 - UserName: Nom d'utilisateur
 - DimServiceName : Nom du service publié par DIM correspondant à cette tâche
 - DimDnsNode : Identifiant de la machine où s'exécute la tâche
- Pour une liste de tâches TaskList :
 - Name : Nom de la liste de tâches
 - Task : Nom de la tâche que cette liste contient
- Pour un nœud ou (une machine) Node:
 - Name : Nom de la machine
 - TaskList : Nom de la liste des tâches s'exécutant dans cette machine
- Pour une liste de nœuds Task :
 - Name : Nom de la liste de nœuds
 - Node : Nom de la machine que cette liste contient

```

<Task>
  <Name>MyTask5</Name>
  <ApplicationName>MyApplication</ApplicationName>
  <Path>dir</Path>
  <Responsable>Chourouk Fanane</Responsable>
  <UserName>Cfanane</UserName>
  <DimServiceName>MyTestDim5</DimServiceName>
  <DimDnsNode>localhost</DimDnsNode>
</Task>

<TaskList>
  <Task>MyTask4</Task>
  <Task>MyTask5</Task>
  <Name>MyTaskList2</Name>
</TaskList>

<TaskList>
  <Name>MyTaskList1</Name>
  <Task>MyTask1</Task>
  <Task>MyTask2</Task>
  <Task>MyTask3</Task>
</TaskList>

<Node>
  <Name>PCLHCB131</Name>
  <MachineName>localhost</MachineName>
  <TaskList>MyTaskList2</TaskList>
  <TaskList>MyTaskList1</TaskList>
</Node>

```

Figure 12: Extrait de l'initial fichier XML créé

- **Extraction de données**

D'après le schéma précédent, on peut noter alors qu'on a utilisé des objets qui contiennent tous les paramètres de l'ensemble des ordinateurs de contrôle.

Pour pouvoir exploiter ces données, il faut pouvoir « **parser** » ces documents. Le parcours du document a été fait grâce au langage Python.

Afin de lire ces données issues du fichier XML, j'ai utilisé l'API DOM [API DOM] de Python avec la bibliothèque PyXML. L'API DOM charge tout le document en mémoire et permet ensuite le parcours manuel grâce à diverses méthodes.

Voici le schéma permettant de visualiser à peu près comment il fonctionne :

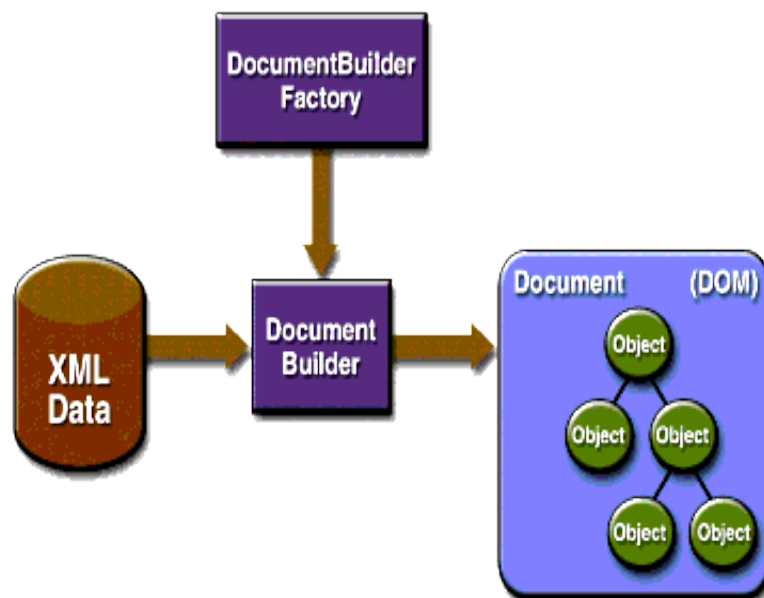


Figure 13: Fonctionnement de L'API DOM

Comme nous pouvons le voir, le parcours du document dépend de deux choses :

- un constructeur de documents (Document Builder),
- une fabrique de constructeurs de documents (Document Builder Factory).

Ensuite il faut parcourir le document depuis sa racine en allant d'élément en élément. Diverses méthodes nous sont alors proposées pour pouvoir récupérer ces éléments et les traiter. DOM peut-être vu comme une arborescence de fichier. Il faut également savoir qu'avec l'API dom de python, tout est élément ou nœud, même le texte. Ainsi, si le <Name> contient "MyTask1" cela voudra dire que le nœud Task de type ELEMENT contient le nœud "MyTask1" de type TEXT.

API permet la manipulation de fichiers XML. Grâce à cette API, on va pouvoir lire le fichier XML(cf Annexe1) pour transformer les noeuds en objets python qu'on aura

définis. On a donc un fichier XML avec des noeuds Task, Node, TaskList, NodeList où chacun de ses nœuds a ses propres attributs. Ces informations seront injectées dans les objets python appropriés qu'on pourra par la suite manipuler.

Prenons par exemple le noeud Node qui représente un ordinateur contrôlé et qui contient des listes de tâches.

Tout d'abord on crée une classe Node (Annexe 1) avec comme attribut le nom du noeud et la liste des tâches de chaque noeud, c'est aussi une autre classe décrite dans l'annexe, ensuite on crée la classe TransformXmlToObjects.

Il s'agit de la classe qui va permettre le traitement du fichier XML. La méthode "`__init__`" fait appel à la méthode "`readXML`" qui va lire le fichier XML grâce à la méthode "`parse`" importée depuis "`xml.dom.minidom`". Cette méthode prend en argument le nom du fichier XML à traiter, c'est-à-dire à le transformer en un objet compréhensible pour python.

Pour la fonction `getNodes()`, Il s'agit de la méthode qui transforme les nœuds nodes en objet Node. Comme il existe plusieurs nodes, nous allons créer une liste dans laquelle on va stocker chaque objet Node créé. En premier lieu on regarde si la liste a déjà été créée et si oui on retourne l'objet déjà existant. Sinon, on crée une nouvelle liste vide.

Ensuite on crée une boucle `for` sur tous les éléments avec comme nom "`node`" contenu dans l'élément root (ici : "`TaskInventory`"). Pour chacun de ces éléments, on regarde s'il s'agit d'un type de noeud ELEMENT. Si oui, on crée un nouvel objet Node et on tente de récupérer le nom de chaque Node en utilisant les méthodes "`getText`" et on appelle ensuite la fonction `addTaskList` qui crée la liste des tâches de chaque nœud. On retourne enfin la liste des nœuds créés.

Ainsi, j'ai développé un ensemble de classe en essayant de les rendre le plus portable possible et un ensemble de méthodes permettant au mieux de traiter les données.

3.2.2 Développement d'un système de contrôle et de supervision

Cette partie a été faite en 2 étapes, d'abord en développant une application appelé `TaskMonitoring` qui contrôle l'état des tâches de chaque ordinateur, et ensuite en développant une autre application nommée `ClusterMonitoring` qui contrôle chaque `TaskMonitoring`, c'est à dire qui vérifie si le contrôle des tâches est effectué.

La figure suivante montre un diagramme de classe UML simplifié de l'architecture de l'application en ne prenant en compte que les principales classes. Ces classes seront traitées dans les deux sous-parties suivantes.

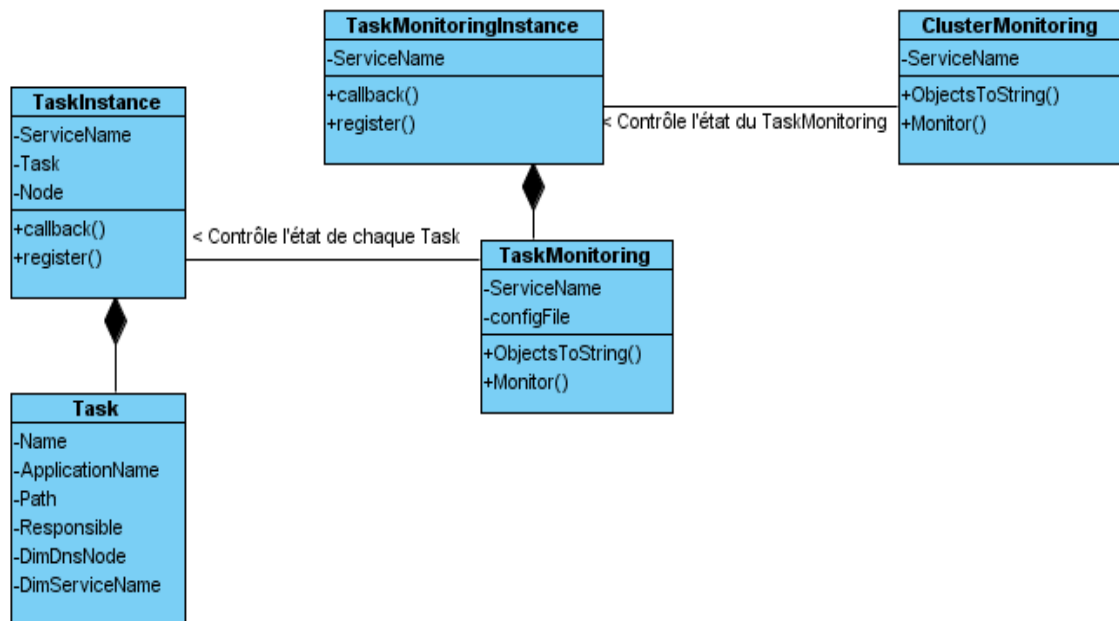


Figure 14: Diagramme de classe de la structure du développement

● L'application TaskMonitoring

Après avoir réalisé la première implémentation de la hiérarchie des données et des tâches à contrôler, ma deuxième tâche était de créer une petite application qui s'inscrit à un service publié par DIM et contrôle l'état de chaque tâche.

La méthode utilisée pour développer l'application est d'avoir recours à des interfaces de communication qui se chargent de traduire les appels vers des fonctions de PyDIM en des appels vers des fonctions de DIM.

Le serveur traite les services envoyés par DIM et publie les résultats sous forme de services. Pour contrôler ces tâches qui sont sous forme des services publiés par DIM, il faut s'inscrire à ces services d'abord et être connecté au même DNS du serveur. Un serveur DIM tourne sur une machine distante permet de répondre à cette question. En lui fournissant le DNS de la machine, il effectue le test et retourne le résultat sous forme d'un service DIM.

474 Servers known - 6840 Services Available Displaying Servers on node ecs03			
DIS_DNS	/ECS03/Minik	/ECS03/coalescence	taskMonitoring2
/ECS03/Logger	/ECS03/out	PVSSSys1Man43:DIMHandler	/ECS03/MyTask4
/ECS03/config_manager	/ECS03/rem	ClusterMonitor	/ECS03/MyTask3
/ECS03/task_manager	/ECS03/IEQ	LHCb_ecs03_Erlog_30579_display	/ECS03/MyTask2
/ECS03/power_manager	/ECS03/irq	LHCb_ecs03_Erlog_30579_history	/ECS03/MyTask1
/ECS03/process_controller	/ECS03/epustat	LHCb_ecs03_Erlog_30579	/ECS03/MyTask3
/ECS03/ib	/ECS03/epustat	PVSSSys109Man30:DIMHandler	/ECS03/MyTask5
/ECS03/eqiq	/ECS03/eqinfo	PVSSSys1Man44:DIMHandler	TDET_File_logger
/ECS03/ps	/ECS03/eqinfo	taskMonitoring	LHCb_File_logger

Figure 15: Exemple des services publiés par DIM

Chaque tâche publiée, veut dire qu'elle est mise à jour, et le but c'est de créer une application qui s'enregistre à ces services publiés par DIM pour savoir si les tâches de chaque ordinateur sont mises à jour .

L'idée était de créer une classe TaskInstance (Annexe2) qui a comme attribut :

- serviceName: le nom du service auquel on va s'abonner
- Task : l'ensemble de tâches qu'on va contrôler,
- node: le nœud correspondant à chaque tâche.

Pour déterminer l'état de chaque tâche, on associe à chacune de ces tâches un timestamp : temps de la dernière mise à jour.

La méthode callback() retourne ce timestamp, ainsi on peut déterminer l'état de chaque tâche en comparant son temps de dernière mise à jour au temps actuel.

Les principales fonctions utilisées de la librairie PyDim dans ce développement sont :

- **dic_info_service()** : pour demander l'abonnement à un service donné.
- **dis_add_service()** : pour créer un nouveau service, elle prend en argument le nom du service qu'on veut créer et son format.
- **dis_start_serving()** : pour démarrer un service.
- **dis_stop_service()** : pour stopper un service.
- **dis_update_service()** : pour publier un service.

De cette manière j'ai pu créer un service appelé «Task Monitoring» qui s'enregistre aux services publiés par DIM et qui publie l'état de chaque tâche de n'importe quel nœud donné.

Pour voir comment cela fonctionne, reportez vous au code commenté fourni en Annexe 2 (classe TaskMonitoring).

L'état de chaque machine prend deux valeurs:

- UP : si la tâche a été mise à jour.
- DOWN : sinon

Voyons maintenant comment les informations sont publiées par ce service:

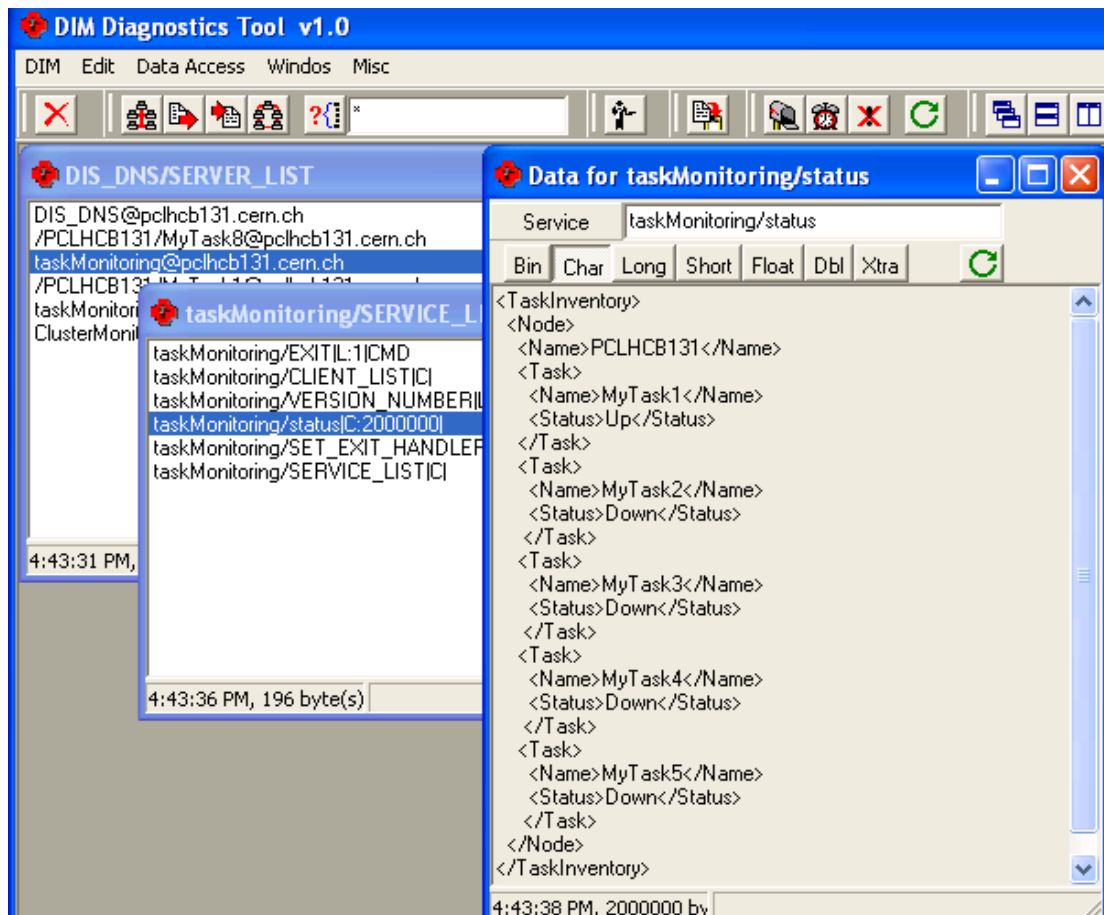


Figure 16: Résultat du premier contrôle publié par DIM

Les informations sont ainsi publiées sous forme d'un string XML. Ce service publie donc l'état et le nom de chaque tâche s'effectuant dans chaque ordinateur contrôlé. Dans ce cas, il y a juste un seul nœud «PCLHCB131».

Après avoir vu comment développer une application qui contrôle les tâches de chaque ordinateur, nous allons aborder dans la partie suivante comment développer une autre application qui contrôle l'application précédente.

• L'application ClusterMonitoring

Comme on l'a vu précédemment, l'application que j'ai développé détermine l'état

des tâches de chaque ordinateur, or, il y aura des ensembles d'ordinateurs à contrôler, donc plusieurs TaskMonitoring.

Pour cela, j'ai développé une autre application ClusterMonitoring; en reprenant plus au moins la même démarche, qui contrôle cette fois-ci l'état des TaskMonitoring au lieu des tâches.

Cette Application permet d'assurer le contrôle des tâches «TaskMonitorings», c'est à dire elle vérifie si le contrôle des tâches est effectué pour enfin envoyer les résultats au PVSS.

Pour cela, j'ai créé un service DIM ClusterMonitoring qui s'enregistre à ces services TaskMonitoring publiés par DIM et récupère l'état de ces services.

Si un contrôle de tâche est effectué alors le service TaskMonitoring est mis à jour, sinon, il ne sera pas publié.

Pour contrôler son état, j'ai créé une classe TaskMonitorInstance(cf Annexe 4) qui a comme attribut le nom des services qui effectuent le contrôle des tâches. Cette classe crée l'ensemble des TaskMonitorings et associe à chacun de ces services un timestamp pour déterminer l'état de chaque application de contrôle.

Tout d'abord, puisque le type des informations publiées par chaque TaskMonitoring est un string XML, je l'ai converti à un format facile à manipuler en PVSS car ce dernier ne contient pas une librairie qui transforme les nœuds XML en objets.

Pour cela, j'ai utilisé encore la librairie DOM (cf Annexe 3) mais en utilisant cette fois-ci les résultats du callback comme un fichier XML.

Ce service ClusterMonitoring publie le nom de tous les services TaskMonitoring auxquels il s'est enregistré, leurs nœuds, leurs tâches, leurs états s'ils sont mis à jour.

Voici ce qu'on obtient en démarrant ce service:

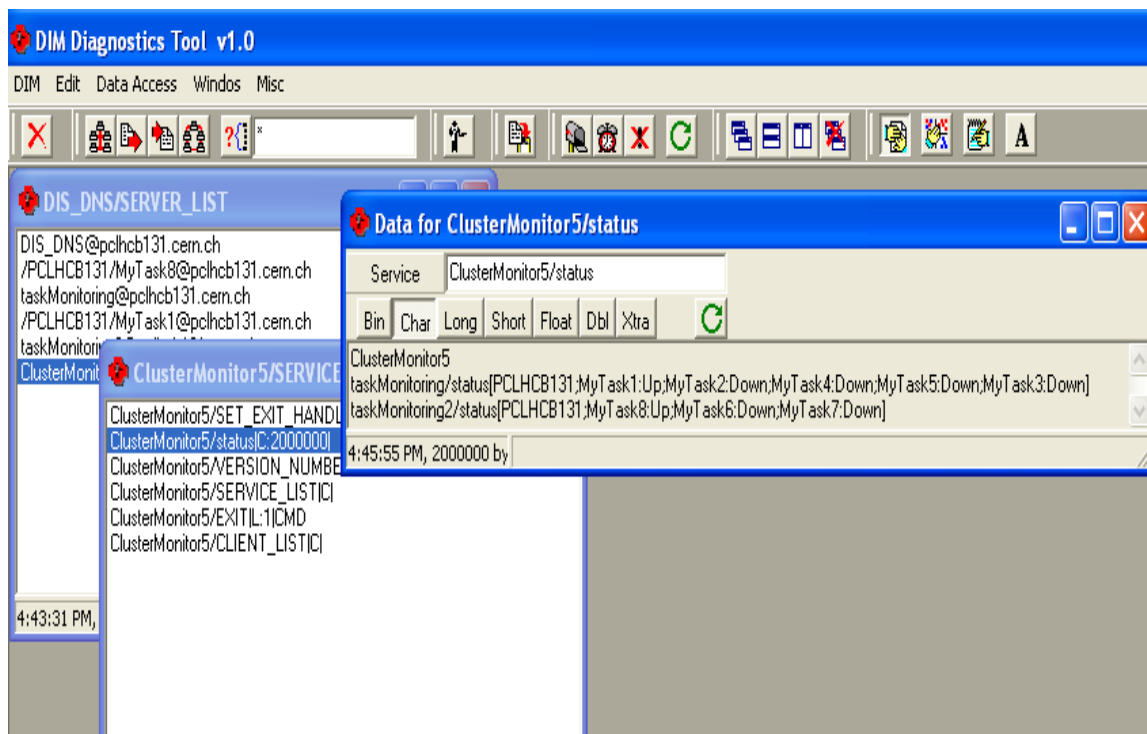


Figure 17: Résultat du 2ème contrôle publié par DIM

Ce service ClusterMonitoring publie donc le nom de tous taskMonitorings et les tâches et machines correspondantes à chaque taskMonitoring. Ces données sont ensuite envoyées sous forme de services au serveur qui effectue leur test.

3.2.3 Création du panneau de supervision en PVSS

Nous arrivons maintenant au dernier module du projet qui consiste à créer un panneau de supervision graphique permettant de visualiser ce contrôle des tâches.

- **Connexion du ClusterMonitoring au PVSS**

DIM établit l'interconnexion entre le client (PVSS) et le serveur (le service créé précédemment). La communication des valeurs entre DIM et PVSS se fait via des DataPoints.

La fonction **fwDim_subscribeService()** permet de connecter un service DIM à un DataPoint en lui donnant en paramètres le nom du service, le nom de DPE ainsi que le nom du manager. Cette interconnexion est expliquée précédemment dans le paragraphe Communication DIM/PVSS).

Sur la figure ci-dessous, le service qui porte le nom ClusterMonitoring/status est relié à un DataPoint, dès que la valeur de ce service change, le DP prend en compte ce changement. Pour établir la communication entre DIM et le serveur j'ai utilisé la librairie FwDIM de fonctions conçue pour DIM.

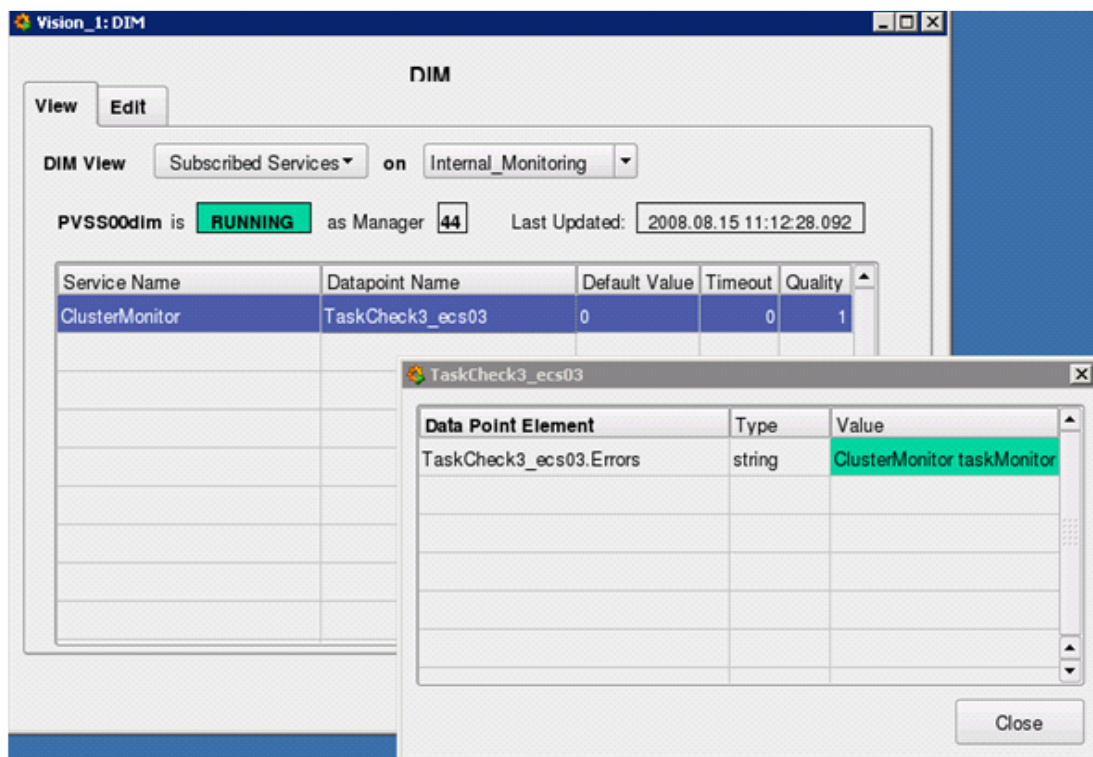


Figure 18: Connexion du service au DataPoint

• Interface graphique

L'élaboration du client PVSS a consisté en deux principales parties qui ont été développées simultanément. La première est la réalisation d'une interface graphique permettant de souscrire aux services DIM. La seconde a consisté à la réalisation d'une bibliothèque de fonctions pour les scripts PVSS, pouvant être facilement réutilisables telles quelles.

Le déroulement du développement du Client a donc été le suivant. Pour chaque nouvelle fonctionnalité :

- Ajout de la fonction adéquate à la bibliothèque
- Réalisation de l'interface graphique, mettant juste en forme derrière des boutons et des zones de saisie des appels aux fonctions de cette bibliothèque.

Il y a donc très peu de code dans les panels PVSS, seulement des vérifications de données et des procédures d'affichage.

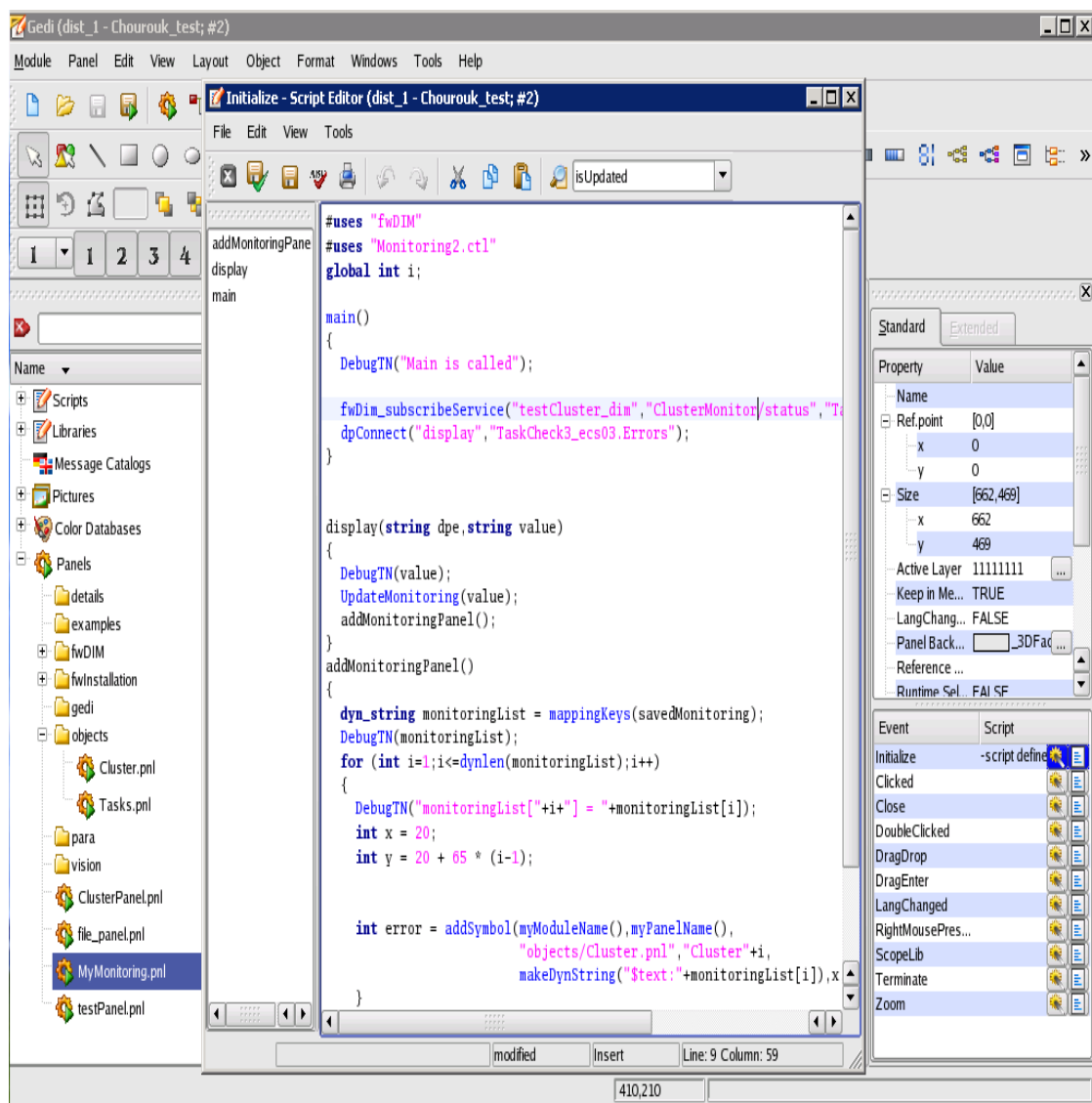


Figure 19: Extrait du script PVSS de la création du panel

Le but étant de visualiser ce contrôle, j'ai développé une interface graphique qui permet de montrer l'état de chaque Monitoring et de toutes ses tâches contrôlées.

L'idée était d'associer à chaque état du TaskMonitoring une couleur, et de montrer ensuite la liste des tâches avec leurs états.

Ainsi, j'ai créé une bibliothèque de 2 fonctions pour mettre à jour le contrôle des tâches et récupérer la couleur à partir de l'état du TaskMonitoring. Ensuite j'ai fait appel à ces fonctions ainsi à des fonctions de la bibliothèque FwDIM dans le script PVSS.

Une interface graphique sous PVSS est un ensemble de boutons, tableaux et zones de texte. Derrière ces objets se trouvent des scripts qui s'exécutent en fonction de l'évènement se produisant sur l'objet.

J'ai utilisé l'ensemble de ces objets pour créer des fenêtres de configuration et de visualisation des tests. Voici à quoi ressemble l'interface graphique de ce contrôle.

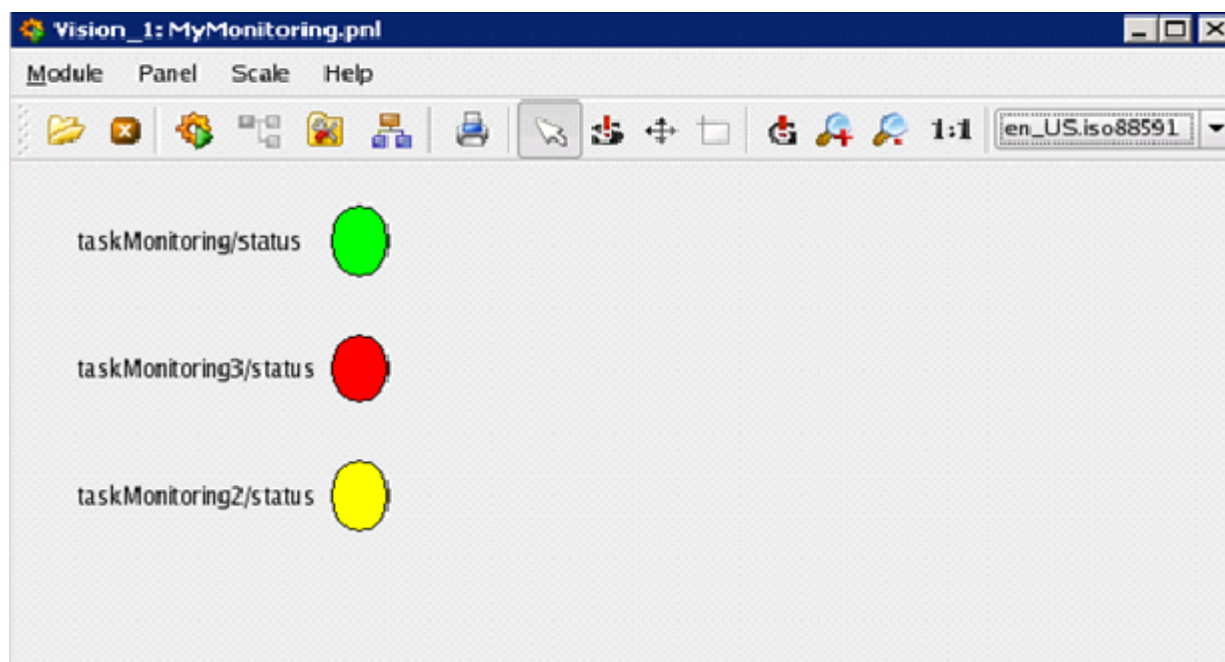


Figure 20: Panneau de supervision des taskMonitorings

Comme nous pouvons le constater, chaque état du TaskMonitoring peut être associé à l'une des 3 couleurs suivantes:

- **Vert** : si le contrôle des tâches « TaskMonitoring » est effectué et si toutes les tâches contrôlées sont mises à jour.
- **Rouge** : si le contrôle des tâches n'est pas effectué.
- **Jaune** : si le contrôle est effectué mais au moins une des tâches contrôlées n'est pas mise à jour.

4 Résultats et discussions

4.1 Résultats

Après avoir vu comment ce projet a été développé, nous allons nous placer du point de vue de l'utilisateur. Lorsque celui-ci sera connecté au terminal du contrôle, il pourra déterminer le nombre des tâches à contrôler, stocker ces informations dans le fichier XML, et puis démarrer les 2 services ClusterMonitoring et TaskMonitoring. Ce dernier peut appeler plusieurs de fois en fonction du nombre de l'ensemble des ordinateurs à contrôler. Enfin, il suffit qu'il active le manager correspondant au pannel développé, et il pourra visualiser ce contrôle. Voici un exemple de cette visualisation:

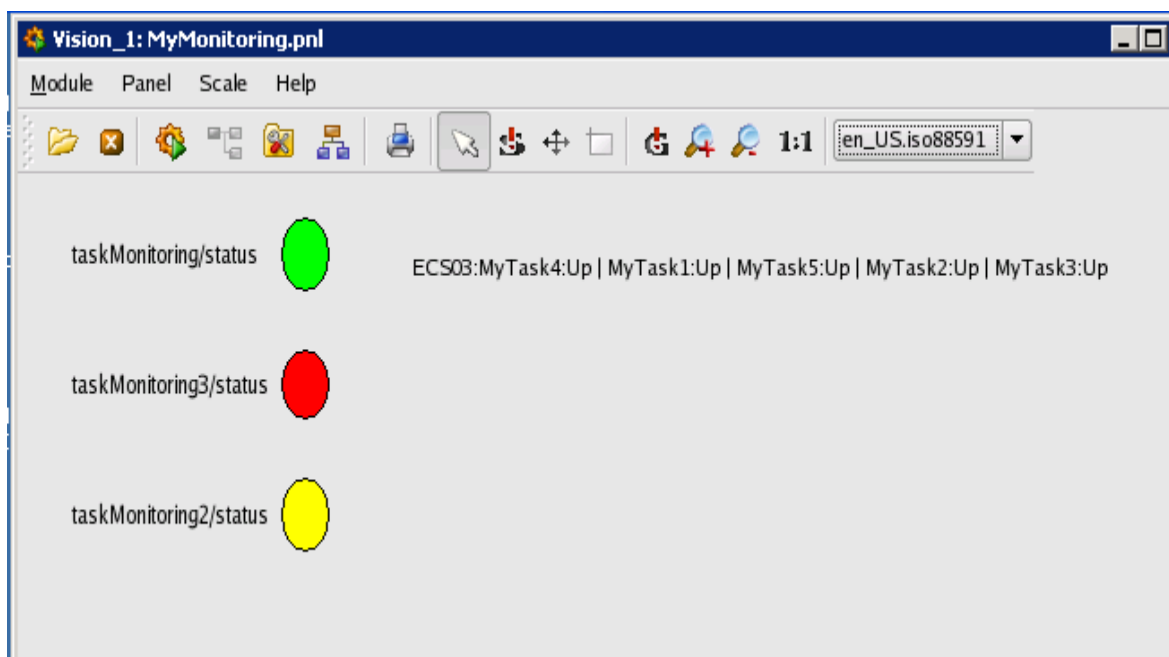


Figure 21: Résultat final du contrôle

Cet outil permet de visualiser aussi l'état de chaque tâche contrôlée en cliquant sur le nom du TaskMonitoring correspondant.

4.2 Enrichissement personnel

Il est indéniable que travailler dans un environnement aussi diversifié, autant sur

le plan des compétences que sur le plan des nationalités, est sans nul doute une expérience humaine extraordinaire. Ce stage m'a permis non seulement de développer mes compétences en langues étrangères et en communication, mais il a aussi développé ma culture scientifique.

La réalisation de ce projet n'a pas été sans difficultés surtout au début, car il fallait avoir une bonne connaissance du langage Python, du principe de DIM, ainsi que la manipulation de PVSS. J'ai appris ainsi à gérer les problèmes techniques et à appréhender de nouveaux concepts de programmation.

Enfin j'ai d'un point de vue plus personnel, beaucoup apprécié ce stage qui m'a permis de découvrir et d'avoir une expérience dans un environnement universel où plusieurs nationalités sont présentes.

Conclusion

Ce stage de cinq mois au sein du CERN m'a permis de mener un projet de bout en bout. J'ai en effet été chargée de développer une application de surveillance dont j'avais l'entière responsabilité, du code jusqu'à l'interface graphique.

Le système de contrôle que j'ai développé répond aux besoins cités dans le cahier de charge, il permet de superviser l'écriture de données lors du stockage des événements produits par le détecteur LHCb. Il est aussi facile à réutiliser vu que chacune des parties développées peut être changée indépendamment.

A l'heure où j'écris ces lignes, le projet n'est pas totalement fini. Si le système de contrôle est réalisé, il reste un certain nombre de petits détails à régler, sur lesquels je compte travailler pendant les trois semaines de stage qu'il me reste. Mais il est fonctionnel et prêt à être intégré dans le système de contrôle de l'expérience LHCb. Les principales difficultés, essentiellement d'ordre technique, rencontrées lors du développement de ce système, ont toutes été surmontées, et n'ont fait qu'approfondir ma technique de codage.

Ce stage m'a aussi beaucoup apporté au niveau personnel. En effet, j'ai eu l'occasion de travailler la communication ainsi que de développer ma culture scientifique à travers des conférences auxquelles j'ai pu assister.

Références bibliographiques

[API DOM] <http://quilovnic.developpez.com/pythondom/>

[DIM] GASPAR Clara DIM
<http://dim.web.cern.ch/dim/>

[DIM/PVSS] GASPAR Clara DIM - PVSS Integration
Disponible sur <http://clara.home.cern.ch/clara/fw/FwDim.html>

[ECS] ECS web page.
<http://lhcb-online.web.cern.ch/lhcb-online/ecs/default.htm>

[PVSS] PVSS
<http://itcobe.web.cern.ch/itcobe/Services/Pvss/Documents/PvssIntro.pdf>

[TDR] Technical Design Report LHCb Online System, Data Acquisition & Experiment Control

ANNEXES

Table des annexes

Annexe 1 : Code de l'extraction de données à partir du fichier XML.....	III
Annexe 2 : Code de la création du TaskMonitoring.....	VI
Annexe 3 : Code de l'extraction de données à partir du résultat du TaskMonitoring	IX
Annexe 4 : Code de la création du ClusterMonitoring.....	XI

Annexe 1 : Code de l'extraction de données à partir du fichier XML

```
import xml.dom.minidom

class NodeList:
    def __init__(self, name):
        self.name = name
        self.nodes = {}

    def addNode(self, node):
        self.nodes[node.name] = node

    def __repr__(self):
        s = "NodeListe %s contains:" % self.name
        for x in self.nodes:
            s += "\n    -> %s" %str(self.nodes[x])
        return s

class Node:
    def __init__(self, name):
        self.name = name
        self.taskLists = {}

    def addTaskList(self, taskList):
        self.taskLists[taskList.name] = taskList

    def __repr__(self):
        s = "Node %s contains:" % (self.name)
        for x in self.taskLists:
            s += "\n    - %s" %str(self.taskLists[x])
        return s

class TaskList:
    def __init__(self, name):
        self.name = name
        self.tasks = {}

    def addTask(self, task):
        self.tasks[task.name] = task

    def __repr__(self):
        s = "TaskList %s contains:" % self.name
        for x in self.tasks:
            s += "\n    -%s" %str(self.tasks[x])
        return s

class Task:
    def __init__(self, applicationName = None,
                  path = None,
                  responsible = None,
                  userName = None,
                  dimServiceName = None,
                  dimDnsNode = None,
                  name = None):
        self.applicationName = None
        self.path = None
        self.responsible = None
        self.userName = None
        self.dimServiceName = None
```

```

        self.dimDnsNode = None
        self.name = None

    def __str__(self):
        s = "Task %s: AppName=%s" %(self.name, self.applicationName)
        return s

    def __repr__(self):
        return str(self)

class TransformXmlToObjects:
    __currentNode__ = None
    __taskLists = None
    __tasks = None
    __nodes = None
    __nodeLists = None

    def __init__(self, xmlfile):
        self.readXml(xmlfile)

    def readXml(self, xmlfile):
        from xml.dom.minidom import parse
        self.doc = parse(xmlfile)

    def getRootElement(self) :
        if self.__currentNode__ == None:
            self.__currentNode__ = self.doc.documentElement
        return self.__currentNode__

    def getTasks(self):
        if self.__tasks != None:
            return self.__tasks
        self.__tasks = {}
        for task in self.getRootElement().getElementsByTagName("Task"):
            if task.parentNode != self.getRootElement():
                continue
            if task.nodeType == task.ELEMENT_NODE:
                t = Task()
                t.applicationName =
self.getText(task.getElementsByTagName("ApplicationName")[0])
                t.path = self.getText(task.getElementsByTagName("Path")[0])
                t.userName =
self.getText(task.getElementsByTagName("UserName")[0])
                t.dimServiceName =
self.getText(task.getElementsByTagName("DimServiceName")[0])
                t.dimDnsNode =
self.getText(task.getElementsByTagName("DimDnsNode")[0])
                t.name = self.getText(task.getElementsByTagName("Name")[0])
                self.__tasks[t.name] = t
            return self.__tasks

    def getText(self, node):
        return node.childNodes[0].nodeValue

    def getTaskLists(self):
        if self.__taskLists != None:
            return self.__taskLists
        self.__taskLists = {}
        for tl in self.getRootElement().getElementsByTagName("TaskList"):
            if tl.parentNode != self.getRootElement():
                continue

```

```

        if tl.nodeType == tl.ELEMENT_NODE:
            name = self.getText(tl.getElementsByTagName("Name")[0])
            taskList = TaskList(name)
            for x in tl.getElementsByTagName("Task"):
                tName = self.getText(x)
                taskList.addTask(self.__tasks[tName])
            self.__taskLists[name] = taskList

    return self.__taskLists

def getNode(self):
    if self.__nodes != None:
        return self.__nodes
    self.__nodes = {}
    for n in self.getRootElement().getElementsByTagName("Node"):
        if n.parentNode != self.getRootElement():
            continue
        if n.nodeType == n.ELEMENT_NODE:
            name = self.getText(n.getElementsByTagName("Name")[0])
            node = Node(name)
            for x in n.getElementsByTagName("TaskList"):
                nName = self.getText(x)
                node.addTaskList(self.__taskLists[nName])
            self.__nodes[name] = node
    return self.__nodes

def getNodeLists(self):
    if self.__nodeLists != None:
        return self.__nodeLists
    self.__nodeLists = {}
    for nl in self.getRootElement().getElementsByTagName("NodeList"):
        if nl.parentNode != self.getRootElement():
            continue
        if nl.nodeType == nl.ELEMENT_NODE:
            name = self.getText(nl.getElementsByTagName("Name")[0])
            nodeList = NodeList(name)
            for x in nl.getElementsByTagName("Node"):
                nlName = self.getText(x)
                nodeList.addNode(self.__nodes[nlName])
            self.__nodeLists[name] = nodeList
    return self.__nodeLists

```

Annexe 2 : Code de la création du TaskMonitoring

```
import time
import dimc
from pydim import *
import sys
from threading import Thread
from time import sleep
import pydim, socket
from pydim import dis_add_service

class TaskInstance:

    def __init__(self, task, node, serviceName):
        self.task = task
        self.node = node
        self.serviceName = serviceName
        self.timestamp = time.time()
        self.register()

    def callback(self, *args):
        print 'Callback: received %s, task %s, node %s' \
            %(args, self.task.name, self.node.name)
        self.timestamp = time.time()

    def register(self):
        print('DIMTASK %s : Registering to service %s'%(self.task,
self.serviceName))
        dic_info_service(self.serviceName, 'C:2000000', self.callback)

class TaskMonitor:
    serviceFormat='C:2000000'
    def __init__(self, serviceName, configFile='tasks.xml', updates=10000):
        self.serviceName = serviceName
        self.updates = updates
        self.parser = taskInventory.TransformXmlToObjects(configFile)
        self.configFile = configFile
        self.tasks = self.parser.getTasks()
        self.taskLists = self.parser.getTaskLists()
        self.nodes = self.parser.getNodes()
        self.nodeLists = self.parser.getNodeLists()

        self.taskInstances = []
        self.svc = dis_add_service(self.serviceName+'/status',
                                TaskMonitor.serviceFormat,
                                self.service,
                                1)

        for nodeName in self.nodes:
            node = self.nodes[nodeName]
            for tlName in node.taskLists:
                tasklist = node.taskLists[tlName]
                for t in tasklist.tasks:
                    task = tasklist.tasks[t]
                    svc = '/' + nodeName.upper() + '/' + t + '/status'
                    print 'Service name:', svc
                    self.taskInstances.append(TaskInstance(task, node, svc))
```

```

        print("DIMTASK %s initialized" %self.serviceName)

def service(self):
    return "status"

def objectsToString(self, taskInstancesUp, taskInstancesDown):
    status = '<TaskInventory>\n'
    err = 0
    for n in self.nodes:
        status += '    <Node>\n'
        status += '        <Name>' + n + '</Name>\n'
        for ti in taskInstancesUp:
            if ti.node.name == n:
                status += '            <Task>\n'
                status += '                <Name>'+ti.task.name+'</Name>\n'
                status += '                <Status>Up</Status>\n'
                status += '            </Task>\n'
        for ti in taskInstancesDown:
            if ti.node.name == n:
                status += '            <Task>\n'
                status += '                <Name>'+ti.task.name+'</Name>\n'
                status += '                <Status>Down</Status>\n'
                status += '            </Task>\n'
            err = err + 1
        status += '    </Node>\n'
    status += '</TaskInventory>\n'
    return status

def monitor(self):
    counter = 0
    TIMEOUT = 10
    UPDATE_INTERVAL = 10

    pydim.dis_start_serving(self.serviceName)
    print('DIMTASK %s : Starting service update'%self.serviceName)
    while counter < self.updates:
        tUp= []
        tDown = []
        for ti in self.taskInstances:
            if (time.time() - ti.timestamp)>TIMEOUT:
                print(" task %s, node %s is not updated" %(ti.task.name,
ti.node.name))
                tDown.append(ti)
            else:
                tUp.append(ti)

        print tUp
        string = self.objectsToString(tUp, tDown)
        print type(string)
        print("DIMTASK : updated %s client " \
            %(pydim.dis_update_service(self.svc, (string, ))))
        sleep(UPDATE_INTERVAL)
        counter += 1
    pydim.dis_stop_serving()

if __name__ == "__main__":

    h = socket.gethostname().upper()

```

```
if len(sys.argv)>1:
    t = TaskMonitor('/'+h+'/'+sys.argv[1])
else :
    t = TaskMonitor('taskMonitoring')
t.monitor()
```

Annexe 3 : Code de l'extraction de données à partir du résultat du TaskMonitoring

```
class Node2:
    def __init__(self, name=None):
        self.name = name
        self.tasks = {}

    def addTask(self, task):
        self.tasks[task.name] = task

class Task2:
    def __init__(self, name = None,
                  status = None):
        self.name = name
        self.status = status

class ParsingXMLString:

    __currentNode__ = None
    __taskList__ = None
    __nodeList__ = None

    def __init__(self, file=None, string=None):
        if not file and not string:
            raise "No input provided"
        if file:
            self.parseXml(file)
        else:
            self.parseString(string)

    def parseXml(self, xmlFile):
        from xml.dom.minidom import parse
        self.doc = parse(xmlFile)

    def parseString(self, xmlString):
        from xml.dom.minidom import parseString
        self.doc = parseString(xmlString)

    def getRootElement(self):
        if self.__currentNode__ == None:
            self.__currentNode__ = self.doc.documentElement
        return self.__currentNode__

    def getNodes(self):
        if self.__nodeList__ != None:
            return
        self.__nodeList__ = []

        for nodes in self.getRootElement().getElementsByTagName("Node"):
            if nodes.parentNode != self.getRootElement():
                continue
            if nodes.nodeType == nodes.ELEMENT_NODE:
                n = Node2()
                n.name = self.getText(nodes.getElementsByTagName("Name")[0])
                for t in nodes.getElementsByTagName("Task"):
                    if t.nodeType == t.ELEMENT_NODE:
                        n.t = self.getTasks(t)
```

```

        n.tasks[n.t] = n.t
        self.__nodeList__.append(n)

    return self.__nodeList__

def getTasks(self, task):

    t = Task2()
    t.name = self.getText(task.getElementsByTagName("Name")[0])
    t.status = self.getText(task.getElementsByTagName("Status")[0])
    return t

def getText(self, node):
    return node.childNodes[0].nodeValue

```

Annexe 4 : Code de la création du ClusterMonitoring

```
import sys
from time import sleep
import pydim, socket
import taskInventory
import datetime
import taskMonitoring
import time
from threading import Thread

class TaskMonitorInstance:

    def __init__(self, serviceName):
        self.xml = ''
        self.serviceName = serviceName
        self.register()
        self.state='DOWN'
        self.timestamp = time.time()
        self.nodes = None

    def callback(self,*args):
        self.timestamp = time.time()
        if len(args)==0:
            print('taskInstance DOWN.')
            self.state='DOWN'
            return
        self.state='UP'
        string = args[0]

        self.parser = ParsingXMLString(string=string)
        self.nodes = self.parser.getNodes()

    def register(self):
        print('DIMTASK : Registering to service %s'%( self.serviceName))
        dic_info_service(self.serviceName, 'C:2000000', self.callback)

class ClusterMonitor:

    serviceFormat='C:2000000'

    def __init__(self, serviceName, updates=10000):
        self.serviceName = serviceName
        self.updates = updates
        self.taskInstances = []
        self.svc = pydim.dis_add_service(self.serviceName+'/status',
                                         ClusterMonitor.serviceFormat,
                                         self.service,
                                         1)
        self.taskInstances.append(TaskMonitorInstance('taskMonitoring/status'))
)

    def service(self):
        return "status"

    def objectsToString(self, monitorOn, monitorOff):
        status = self.serviceName
```

```

for ti in monitorOff:
    status += ' '+ti.serviceName
for ti in monitorOn:
    status += ' '+ti.serviceName
    if not ti.nodes:
        monitorOff.append(ti)
        continue
    for node in ti.nodes:
        status += '['+node.name
        for task in node.tasks:
            status += ';' +task.name
            status += ':' +task.status
        status += ']'
return status

def monitor(self) :
    TIMEOUT = 10
    counter = 0
    pydim.dis_start_serving(self.serviceName)
    while counter < self.updates:
        tUp= []
        tDown = []
        for ti in self.taskInstances:
            if (time.time() - ti.timestamp)>TIMEOUT:
                print ('%s is not updated'%ti.serviceName)
                values = 'DOWN'
                tDown.append(ti)
            else:
                values = 'UP'
                tUp.append(ti)
        print values
        string = self.objectsToString(tUp, tDown)
        print('Updated %d clients' \
            %(pydim.dis_update_service(self.svc, (string, )))
        )
        sleep(1)

if __name__ == "__main__":
    h = socket.gethostname().upper()
    if len(sys.argv)>1:
        t = ClusterMonitor('/'+h+'/' +sys.argv[1])
    else :
        t = ClusterMonitor('ClusterMonitoring')
    t.monitor()

```