



Institut **S**upérieur d'Informatique
de **M**odélisation et de leurs **A**pplications

Complexe des Cézeaux
BP 125 – 63173 Aubière Cedex, FRANCE



Organisation **E**uropéenne
pour la **R**echerche **N**ucléaire

CERN
CH-1211 Genève 23, SUISSE

Rapport de stage de 2^{ème} année
Filière 3

Supervision des clusters de surveillance et de traitement de données dans l'expérience LHCb

Présenté par: **Raúl DURÁN BRAVO**
Responsable CERN: **M. Markus FRANK**
Responsable ISIMA : **M. Emmanuel MESNARD**

Du 14 avril au
12 septembre
2008



Institut **S**upérieur d'Informatique
de **M**odélisation et de leurs **A**pplications

Complexe des Cézeaux
BP 125 – 63173 Aubière Cedex, FRANCE



Organisation **E**uropéenne
pour la **R**echerche **N**ucléaire

CERN
CH-1211 Genève 23, SUISSE

Rapport de stage de 2^{ème} année
Filière 3

Supervision des clusters de surveillance et de traitement de données dans l'expérience LHCb

Présenté par: **Raúl DURÁN BRAVO**
Responsable CERN: **M. Markus FRANK**
Responsable ISIMA : **M. Emmanuel MESNARD**

Du 14 avril au
12 septembre
2008

Remerciements

Je remercie mon tuteur d'entreprise M. Markus FRANK qui m'a aidé énormément depuis mon arrivée et pendant tout le stage. Je le remercie pour m'avoir confié un travail très important et fournit tout ce dont j'ai eu besoin pour le réaliser, ainsi que pour m'avoir fait des remarques pertinentes et m'avoir guidé pendant le déroulement de mon séjour. De plus il m'a donné l'opportunité de travailler avec lui et l'équipe de travail du LHCb en me recrutant.

Je tiens également à remercier aux MM Niko Neufeld et Radu Stoica qui ont répondu à mes doutes et m'ont orienté durant la réalisation de mon travail.

Aussi je remercie M. Emmanuel MESNARD mon tuteur de l'ISIMA pour sa visite et ses conseils.

Et en dernier mais non par ordre d'importance je souhaite exprimer toute ma gratitude à mes amis de CarpeDiem qui ont rendu l'expérience d'être au CERN et à Genève fantastique.

Table de figures

Figure 1: Le tunnel du LHC.....	11
Figure 2: Le détecteur LHCb	12
Figure 3: Fonctionnement de DIM.....	13
Figure 4: Le système de surveillance du LHCb.....	14
Figure 5: Interaction des besoins du projet identifiés.....	16
Figure 6: Types de nœuds.....	18
Figure 7: Listes de tâches.....	19
Figure 8: Listes de tâches par type de nœud.....	19
Figure 9: Les TaskSupervisor dans les clusters	20
Figure 10: Le diagramme de classes du TaskSupervisor.....	21
Figure 11: Analyse Grammaticale des fichiers XML pour le TaskSupervisor.....	21
Figure 12: Lancement des NodeMonitor.....	22
Figure 13: Création de service du TaskSupervisor.....	23
Figure 14: Souscription de NodeMonitor au service ps/data.....	23
Figure 15: Chaîne de caractères de ps/data	23
Figure 16: Boucle principale de NodeMonitor	24
Figure 17: Structure TmTask.....	24
Figure 18: Format d'information en XML	25
Figure 19: Publication d'information du TaskSupervisor	26
Figure 20 : Le ClusterMonitor dans le système de surveillance	27
Figure 21: Le diagramme de classes du ClusterMonitor	28
Figure 22: Analyse Grammaticale des fichiers XML pour le ClusterMonitor.....	29
Figure 23: Souscription au service DIS_DNS/SERVER_LIST.....	29
Figure 24: Chaînes de caractères du service DIS_DNS/SERVER_LIST	29
Figure 25: Séparation de chaînes dans le ClusterCollector	30
Figure 26: La fonction show()	31
Figure 27: Démarrage du TaskSupervisor et création des NodeMonitor.....	32
Figure 28: Contenu du service TaskSupervisor (affichage en DIM).....	33
Figure 29: Contenu du service TaskSupervisor (affichage à l'écran).....	33
Figure 30: Contenu du service ClusterCollector (affichage en DIM)	34
Figure 31: Contenu du service ClusterCollector (affichage à l'écran)	35
Figure 32: TaskSupervisor et ClusterCollector au même temps	36

Résumé

Le LHC au CERN est le projet qui cherche les réponses aux différentes questions que la physique se pose, comme la présence du boson de Higgs, l'origine des masses des particules, l'existence de la supersymétrie, et bien d'autres.

Dans l'expérience LHCb se feront les analyses en relation aux asymétries de la matière et l'antimatière. Les collisions produiront une information éphémère qui sera obtenue par un système de capture qui enverra les données aux fermes de traitement et de surveillance. Une fois prétraitée, l'information pertinente est sélectionnée pour être stockée de manière permanente.

Mon travail a été de développer des outils qui permettent de superviser les systèmes de surveillance et de traitement, trouver leurs erreurs et les notifier au groupe de travail.

La ferme de traitement compte 50 clusters d'environ 1000-2000 nœuds et la ferme de surveillance possède un cluster de 50 nœuds.

Les applications que j'ai faites sont le TaskSupervisor et le ClusterMonitor. Le TaskSupervisor trouve les erreurs dans chaque nœud d'un cluster et affiche l'information pendant que le ClusterMonitor collectionne l'information des TaskSupervisor pour la concentrer dans un seul endroit.

Mots clés : Supervision, cluster, nœud, surveillance, ferme de traitement, notification d'erreurs.

Abstract

The LHC at CERN is the project that looks for answers to different questions that the physics has made itself, such as the presence of the Higgs boson, the origin of mass in particles, the existence of supersymmetry, and many others.

In the LHCb experiment, analysis corresponding to the asymmetry between matter and antimatter will be made. The collisions will produce ephemeral information that will be obtained by a readout system that will send the data to the processing and monitoring farms. Once preprocessed, the relevant information will be selected to be permanently stored.

My work was to develop tools that permit the supervision of the monitoring and processing systems, find their errors and notify the workgroup.

The processing farm has 50 clusters with approximately 1000-2000 nodes and the monitoring farm has a cluster with 50 nodes.

The applications I made are the TaskSupervisor and the ClusterMonitor. The TaskSupervisor finds the errors in each node of a cluster and publishes the results while the ClusterMonitor collects the information of the TaskSupervisors and concentrates them in a single place.

Keywords: Supervision, cluster, node, monitoring, data processing farm, error notification.

Abréviations

ALICE : A Large Ion Collider Experiment

ATLAS : A Toroidal LHC ApparatuS

CERN : Organisation Européenne pour la Recherche Nucléaire

CMS : Compact Muon Solenoid

DIM : Distributed Information Management System

LHC : Large Hadron Collider

LHCb : Large Hadron Collider beauty

XML : eXtensible Markup Language

Table de matières

Remerciements.....	3
Table de figures	4
Résumé	5
Abstract	6
Abréviations.....	7
Table de matières	8
Introduction.....	9
Chapitre 1 : Présentation générale.....	10
1.1 Histoire.....	10
1.1.1 CERN (Organisation Européenne pour la Recherche Nucléaire).....	10
1.1.2 LHC (Large Hadron Collider)	11
1.1.3 LHCb (Large Hadron Collider beauty).....	12
Chapitre 2: Étude de cas.....	14
2.1 Introduction: Le système de surveillance	14
2.2 Analyse du problème	15
2.3 Solution Choisi	16
2.3.1 Les listes	16
2.3.2 Le TaskSupervisor.....	17
2.3.3 Le ClusterMonitor	17
Chapitre 3: Réalisation de la solution.....	18
3.1 Définition des fichiers XML	18
3.1.1 NodeList (liste de nœuds)	18
3.1.2 TaskList (liste de tâches).....	19
3.2 L'implémentation du TaskSupervisor	20
3.2.1 Définition.....	20
3.2.2 Le diagramme de classes.....	20
3.2.3 Démarrage du TaskSupervisor	21
3.2.4 Obtention et Traitement de l'information (NodeMonitor)	22
3.2.5 Publication d'erreurs.....	25
3.3 ClusterMonitor.....	27
3.3.1 Définition.....	27
3.3.2 Diagramme de classes	28
3.3.3 Démarrage du ClusterMonitor	29
3.3.4 Obtention et traitement de l'information	29

3.3.5 Publication d'erreurs	31
Chapitre 4: Résultats.....	32
4.1 Résultats du TaskSupervisor	32
4.2 Résultats du ClusterMonitor	34
Conclusion	37
Annexe A: Code source du TaskSupervisor.....	I
Annexe B: Code source du ClusterMonitor	XIV

Introduction

L'Organisation Européenne pour la Recherche Nucléaire (CERN) a créé le LHC qui est l'accélérateur de particules le plus puissant du monde et sera mis en opération le 10 septembre 2008. Le LHC se trouve dans un tunnel de 27 km au dessous de la frontière franco-suisse.

Dans les différents points du tunnel, des rayons de particules entreront en collision presque à la vitesse de la lumière, et presque au zéro absolu pour reproduire les conditions du big bang. Chaque point du tunnel étudiera des phénomènes différents.

Le LHCb explorera les différences entre la matière et l'antimatière en étudiant un type de particule appelée «quark beauté» ou «quark b».

Pour accomplir les objectifs de l'étude, un grand système de surveillance de 50 clusters qui contient environ 2000 nœuds capturera les résultats des collisions.

Mon travail fait partie du développement d'outils de supervision pour le système de surveillance. Plus précisément la création des programmes qui permettent de trouver des erreurs à un niveau d'exécution du processus dans les clusters et qui puissent en informer l'équipe de travail rattachée de l'expérience.

Le rapport se déroulera en quatre parties: la première fera une présentation générale du projet, la seconde décrira l'étude de cas, ensuite une troisième partie présentera la réalisation de la solution et finalement une dernière partie traitera des résultats du projet.

Chapitre 1 : Présentation générale

1.1 Histoire

1.1.1 CERN (Organisation Européenne pour la Recherche Nucléaire)

En 1952, un ensemble de 11 gouvernements européens décident de créer un Conseil Européen pour la Recherche Nucléaire (CERN) dans le petit village de Meyrin dans le canton de Genève, près de la frontière franco-suisse.

Le 29 septembre 1954, la convention du CERN est ratifiée par 12 États européens, le CERN est officiellement créé et se nomme maintenant Organisation Européenne pour la Recherche Nucléaire.

Aujourd'hui, le CERN emploie environ 2500 personnes. Le personnel scientifique et technique du laboratoire conçoit et construit les accélérateurs de particules et assure leur bon fonctionnement. Il contribue également à la préparation et à la mise en œuvre des expériences scientifiques complexes, ainsi qu'à l'analyse et à l'interprétation des résultats.

Environ 8000 scientifiques visiteurs, soit la moitié des physiciens des particules du monde, viennent au CERN pour faire des recherches. 580 universités et 85 nationalités sont représentées.

1.1.2 LHC (Large Hadron Collider)

Le Grand Collisionneur de Hadrons au CERN (Figure 1: Le tunnel du LHC) sera le projet scientifique le plus ambitieux jamais réalisé. Les résultats des expériences du LHC cherchent à répondre des questions que la physique s'est posé depuis longtemps.

Le LHC est situé dans un anneau de 27 km enterré à la campagne près Genève dans la frontière franco-suisse entre le lac Léman et le Jura. Le LHC, qui commencera à fonctionner le 10 septembre 2008 est le collisionneur de particules le plus puissant de l'histoire. Protons de haute énergie circuleront dans les rayons au sens contraire les unes des autres et seront écrasés pour chercher des traces de supersymétrie, matière noire et les origines de la masse.

Les rayons sont faits de bouquets qui contiennent milliards de protons. Ils seront accélérés presque à la vitesse de la lumière et injectés au LHC où ils continueront en circulant pendant plusieurs heures guidés par milles d'aimants superconducteurs très puissants pour entrer en collision dans le cœur des principales expériences, connues par leurs acronymes : ALICE, ATLAS, CMS et LHCb. Les détecteurs observeront très attentivement pendant que l'énergie résultant des collisions de protons se transforme fugacement dans une pléthore de particules exotiques.

Les détecteurs pourront voir jusqu'à 600 millions de collisions par second, avec les expériences qui sélectionneront les données pour vestiges d'événement extrêmement rares comme la création de l'amplement recherché boson de Higgs.

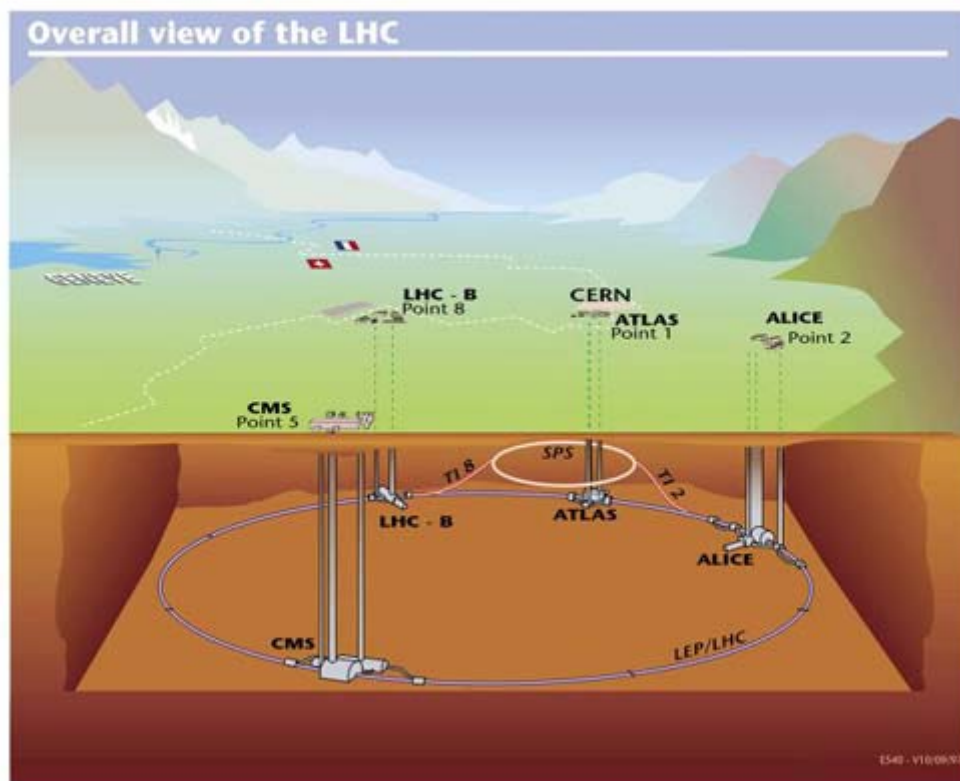


Figure 1: Le tunnel du LHC

1.1.3 LHCb (Large Hadron Collider beauty)

L'expérience LHCb cherche à comprendre pourquoi nous vivons dans un univers qui semble être constitué entièrement de matière, sans aucune présence d'antimatière.

L'expérience explorera les différences entre matière et antimatière en étudiant un type de particule appelée «quark beauté» ou «quark b». Le LHC recréera les instants juste après le Big Bang, pendant lesquels les paires de quarks b et d'antiquarks b auraient été produites.

LHCb 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.

Une grande variété de types de quarks seront créés par le LHC avant de se désintégrer rapidement pour former d'autres particules. Pour intercepter les quarks b, la collaboration LHCb a mis au point des trajectographes mobiles, installés au plus près de la trajectoire des faisceaux.

La collaboration LHCb compte 650 scientifiques représentant 48 instituts et 13 pays (avril 2006).

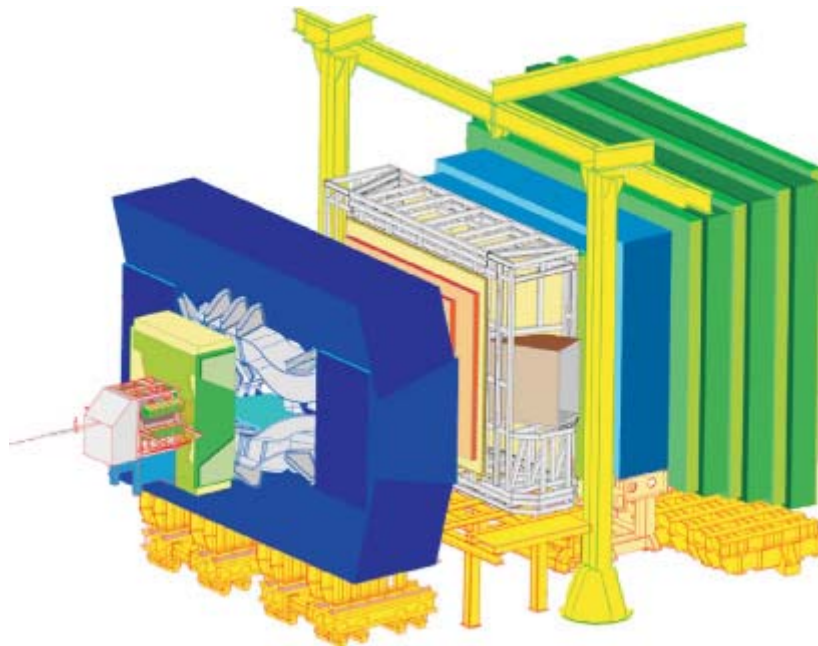


Figure 2: Le détecteur LHCb

Le détecteur LHCb (Figure 2)

- Dimensions : 21 mètres de long, 13 mètres de large et 10 mètres de haut
- Poids : 5600 tonnes
- Configuration : spectromètre à petits angles avec détecteurs planaires

Situation : Ferney-Voltaire, France.

1.2 Outils de travail

1.2.1 Python

Python est un langage de programmation orienté objet, multiplateforme et avec une licence de libre. Python a été choisi dû à sa facilité pour manipuler des chaînes de caractères, syntaxe simple et facilité de maintenance. Des bibliothèques Python dédiées à l'interaction avec DIM ont été déjà développées.

1.2.2 XML

XML (eXtensible Markup Language) est un standard de représentation de données très utilisé pour l'échange d'information. On lui a choisi pour sa facilité pour structurer des données, la facilité d'utilisation entre plateformes et l'universalité du format.

On utilisera le standard XML pour le format de l'information des fichiers maîtres, et pour la sortie en écran et comme la sortie du service de l'information produite pour les applications de surveillance.

1.2.3 DIM (Distributed Information Management System) ^[1]

DIM est un système développé pour des autres expériences du CERN et qui s'utilise maintenant pour le LHCb. DIM permet aux processus du système de surveillance de se communiquer entre eux, même s'ils ne savent pas où sont les autres processus.

DIM se base dans le paradigme client/serveur et son concept basic est le «service». Les serveurs fournissent des services aux clients. Un service est normalement une information (quelque soit le type ou la taille) et est reconnu par un nom.

Le fonctionnement basic de DIM est le suivant (Figure 3):

Les serveurs publient ses services en les enregistrant avec le nom du serveur (normalement au démarrage). Les clients se souscrivent aux services en demandant au serveur de noms quel serveur fournit le service demandé, après, ils demandent directement au service avec les paramètres nécessaires.

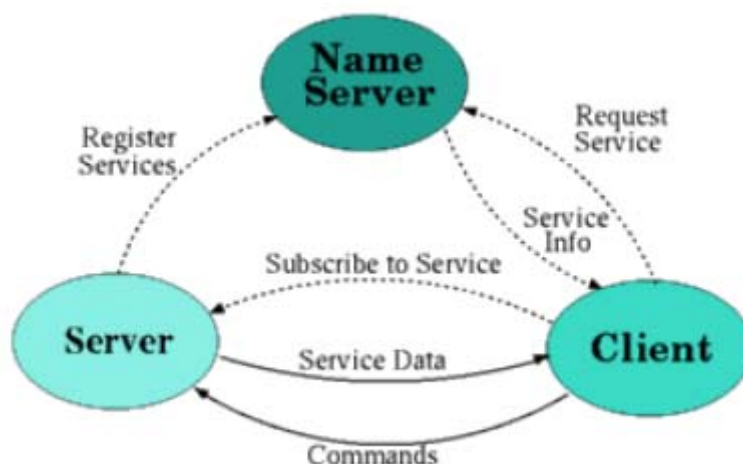


Figure 3: Fonctionnement de DIM

Chapitre 2: Étude de cas

2.1 Introduction: Le système de surveillance ^{[2] [3] [4]}

Dans la Figure 4 on peut apprécier le fonctionnement général du système de surveillance. D'abord les collisions auront lieu dans le détecteur LHCb (partie haute) où le système de capture de niveau électronique (readout network) enverra les données aux fermes de traitement HLT (les 50 clusters) et surveillance (le cluster de 50 nœuds) qui sélectionneront les événements pertinents pour leur envoyer au stockage de longue durée (le cluster de 4 nœuds) pour faire des analyses postérieures hors de ligne.

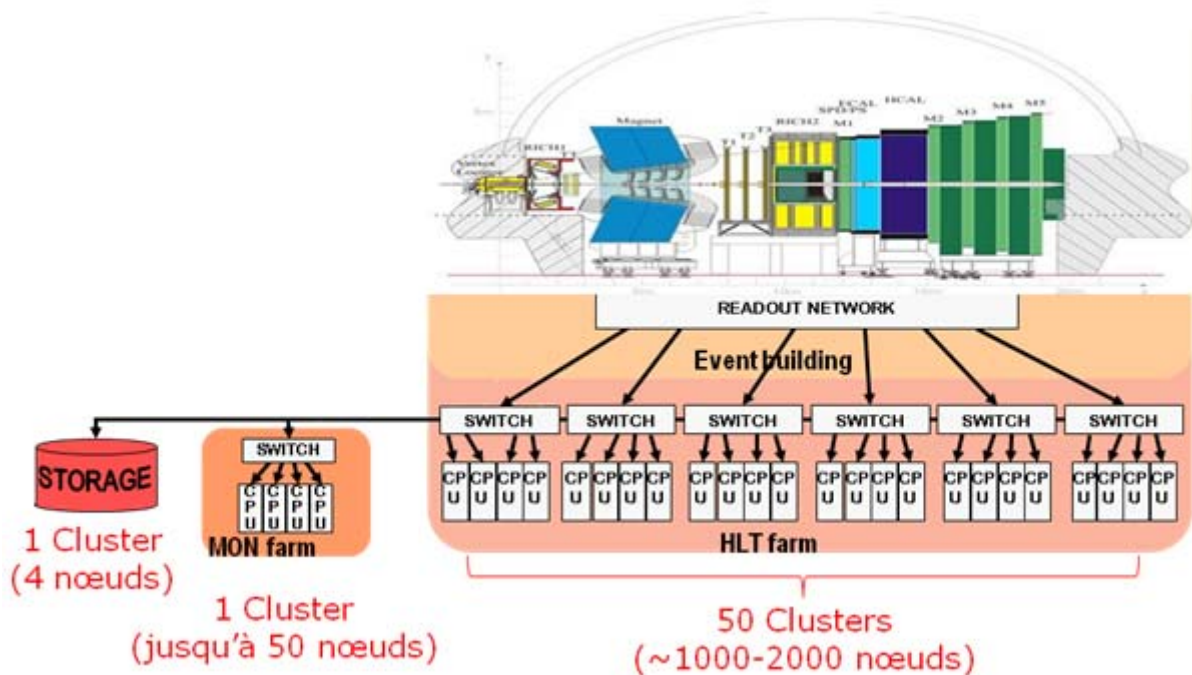


Figure 4: Le système de surveillance du LHCb

Dans les différentes fermes il y a un total d'environ 2000 processeurs rangés en 50 sous-fermes de jusqu'à 40 nœuds chacune. Les nœuds sont connectés au système de control qui réunit aussi toute l'information de la surveillance et au même temps, les nœuds ont des services qui s'exécutent et publient l'information.

2.2 Analyse du problème

L'expérience LHCb a plusieurs outils qui aident à la supervision du système de surveillance et qui sont cruciaux pour détecter les possibles problèmes et ne pas rater l'opportunité de détecter information précieuse pour l'expérience.

Le système de surveillance acquiert les données de l'expérience LHCb, pendant que les outils de supervision doivent s'assurer que le système de surveillance marche correctement.

C'est ici où l'opportunité de développement d'outils se présente parce qu'on manque plus d'outils pour surveiller les clusters et leurs nœuds au niveau des processus qui s'exécutent et maintenir les erreurs au minimum.

Afin d'exécuter tous les processus qui participent dans ces activités, plusieurs facteurs doivent être pris en compte, en étant les plus importants d'entre eux que le réseau soit proprement configuré et que l'infrastructure de tâches fonctionne correctement.

Par exemple, s'il y a un problème électrique et un cluster s'éteint, comment on peut savoir que le cluster n'est pas actif si nous ne sommes pas à côté de lui pour nous rendre compte ?

Qu'est-ce qui se passe si un nœud a été juste redémarré pour quelque soit la raison ?, comment on sait si toutes les tâches nécessaires ont été mis en marche ?

Le projet à réaliser est créer et développer un système qui permettra vérifier que :

- Tous les nœuds fonctionnent.
- Toutes les tâches sont dans tous les nœuds.
- En cas de fautes, le système de control de l'expérience doit être informé.
- L'affichage d'états doit être développé pour permettre une investigation rapide et savoir :
 - o Où la faute a eu lieu.
 - o C'est quoi qui échoue.

2.3 Solution Choisi

Pour vérifier le fonctionnement du réseau de nœuds et clusters, et après plusieurs réunions avec le groupe de travail, on a décidé implémenter une solution en trois points:

- Les listes
- Le TaskSupervisor
- Le ClusterMonitor

À grosso modo le fonctionnement attendu est le suivant (Figure 5) :

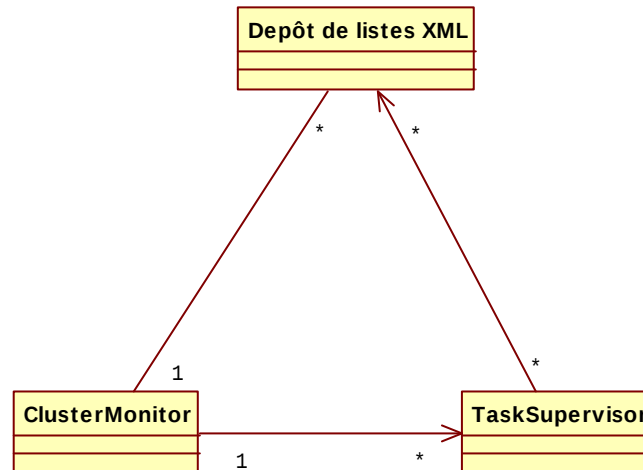


Figure 5: Interaction des besoins du projet identifiés

Les listes seront disponibles pour le démarrage des deux applications, les TaskSupervisor seront en charge de superviser les nœuds de chaque cluster et à la fois le un seul ClusterMonitor supervisera tous les TaskSupervisor

2.3.1 Les listes

Comme il y a beaucoup de nœuds à surveiller et plein de tâches dans chaque nœud, il faut créer et définir des fichiers qui contiendront les listes maîtresses que les applications utiliseront pour faire leur travail, les fichiers sont :

- La liste de nœuds «NodeList»
- La liste de tâches «TaskList»

La liste de nœuds ou «NodeList» contiendra un index des nœuds qui existent en chaque cluster et le type de liste de tâches liée à chaque nœud. Il y aura une liste de nœuds différente pour chaque cluster.

La liste de tâches ou «TaskList» contiendra les tâches essentielles qui doivent être exécutées en chaque nœud.

Le format des fichiers sera XML et tous seront dans un dépôt commun. Le contenu et le format doivent être définis.

2.3.2 Le TaskSupervisor

Le TaskSupervisor est l'application qui publiera «l'état de santé» de chaque nœud en comparant sa liste de tâches contre les tâches qui s'exécutent.

Les fonctions du TaskSupervisor seront :

- Vérifier l'état des nœuds.
- Comparer la liste de tâches contre les tâches qui s'exécutent.
- Publier les résultats.

2.3.3 Le ClusterMonitor

Le ClusterMonitor est l'outil qui réunira l'information de chacun des TaskSupervisor dans un seul lieu.

Les fonctions principales du ClusterMonitor seront :

- Vérifier l'état des clusters
- Comparer la liste de nœuds contre les nœuds ACTIFS.
- Concentrer les données de tous les TaskSupervisor.
- Publier les résultats.

Chapitre 3: Réalisation de la solution

3.1 Définition des fichiers XML

Comme on a fait noter dans l'analyse du problème, on avait besoin de deux fichiers maîtres qui seront utilisés pour les applications à développer.

Les applications auront avec les fichiers un point de départ pour faire les comparaisons et savoir quels sont tous les nœuds qui doivent être actifs et les tâches essentielles de chaque nœud. Les listes doivent avoir une relation entre eux.

Ces deux fichiers sont la liste de nœuds et la liste de tâches.

3.1.1 NodeList (liste de nœuds)

La liste de nœuds sert à avoir l'index des nœuds dans un cluster, mais en ayant différents types de nœuds, la liste doit servir pour distinguer quel type de nœud est chacun et doit contenir une façon de la lier à la liste de tâches pour savoir quelles tâches correspondront à chaque nœud.

On va prendre le cluster MONA08 pour expliquer la façon de définir le fichier XML avec la liste.

On a défini un élément appelé NodeList dans le fichier XML qui contiendra le nom du cluster et les différents nœuds libellés comme « Node », à chaque nœud on a ajouté comme des attributs le nom du nœud, et le type de nœud.

Le type de nœud est un attribut très important parce qu'avec lui on établit la différence entre les nœuds qu'on va utiliser après pour appeler les listes de tâches qui correspondent. (Figure 6)

Les deux types de nœuds principaux sont le « Nœud de Surveillance » et le « Nœud de Travail ». Alors on a utilisé MonitoringControl et MonitoringWorker, comme définition des nœuds pour l'exemple et bien sûr on peut utiliser plus selon les besoins futures.

```
<NodeList>
  <Name>MONA08</Name>
  <Node name="mona08" type="MonitoringControl"/>
  <Node name="mona0801" type="Monitoringworker"/>
  <Node name="mona0802" type="Monitoringworker"/>
  <Node name="mona0803" type="Monitoringworker"/>
  <Node name="mona0804" type="Monitoringworker"/>
  <Node name="mona0805" type="Monitoringworker"/>
</NodeList>
```

Figure 6: Types de nœuds.

Après, le type de nœud nous servira pour assigner une ou plusieurs listes de tâches à chaque nœud selon leurs activités. On se réfère au type de nœud et on ajoute les listes de tâches désirées

Dans la Figure 7 on peut apprécier la façon d'utiliser le type de nœud pour appeler les différentes listes de tâches nécessaires.

```

<NodeType>
  <Name>MonitoringControl</Name>
  <TaskList>StandardMonitoring</TaskList>
  <TaskList>MonitoringControl</TaskList>
</NodeType>
<NodeType>
  <Name>Monitoringworker</Name>
  <TaskList>StandardMonitoring</TaskList>
  <TaskList>Monitoringworker</TaskList>
</NodeType>

```

Figure 7: Listes de tâches.

MonitoringControl et MonitoringWorker deviennent les noms pour lesquels on va reconnaître un groupe de listes de tâches associées à un nœud.

3.1.2 TaskList (liste de tâches)

La liste de tâches est le fichier où on a fait référence aux tâches essentielles qui sont associées à chaque nœud selon le type de nœud.

La Figure 8 est un exemple qu'on a utilisé pour les premiers nœuds testés. On a défini dans le fichier trois éléments TaskList avec un nom et quelques tâches.

Le nom de la liste de tâches est la liaison avec la liste de nœuds. Dans ce cas ils sont StandardMonitoring, MonitoringWorker et Monitoring Control.

```

<TaskList>
  <Name>StandardMonitoring</Name>
  <Task>/&lt;Node&gt;/Monitoring_Slice00Srv</Task>
  <Task>/&lt;Node&gt;/Monitoring_Slice01Srv</Task>
  <Task>/&lt;Node&gt;/Monitoring_Slice02Srv</Task>
  <Task>/&lt;Node&gt;/Monitoring_Slice03Srv</Task>
</TaskList>

<TaskList>
  <Name>Monitoringworker</Name>
  <Task>/&lt;Node&gt;/Dummy_Service</Task>
</TaskList>

<TaskList>
  <Name>MonitoringControl</Name>
  <Task>/&lt;Node&gt;/MoniCONTROL</Task>
</TaskList>

```

Figure 8: Listes de tâches par type de nœud.

3.2 L'implémentation du TaskSupervisor

3.2.1 Définition

Le TaskSupervisor est l'application qui détermine la santé de chaque nœud, il obtiendra toute l'information vitale des nœuds d'un cluster en les surveillant et publiera les comportements anormaux.

Il y a un TaskSupervisor par cluster (Figure 9).

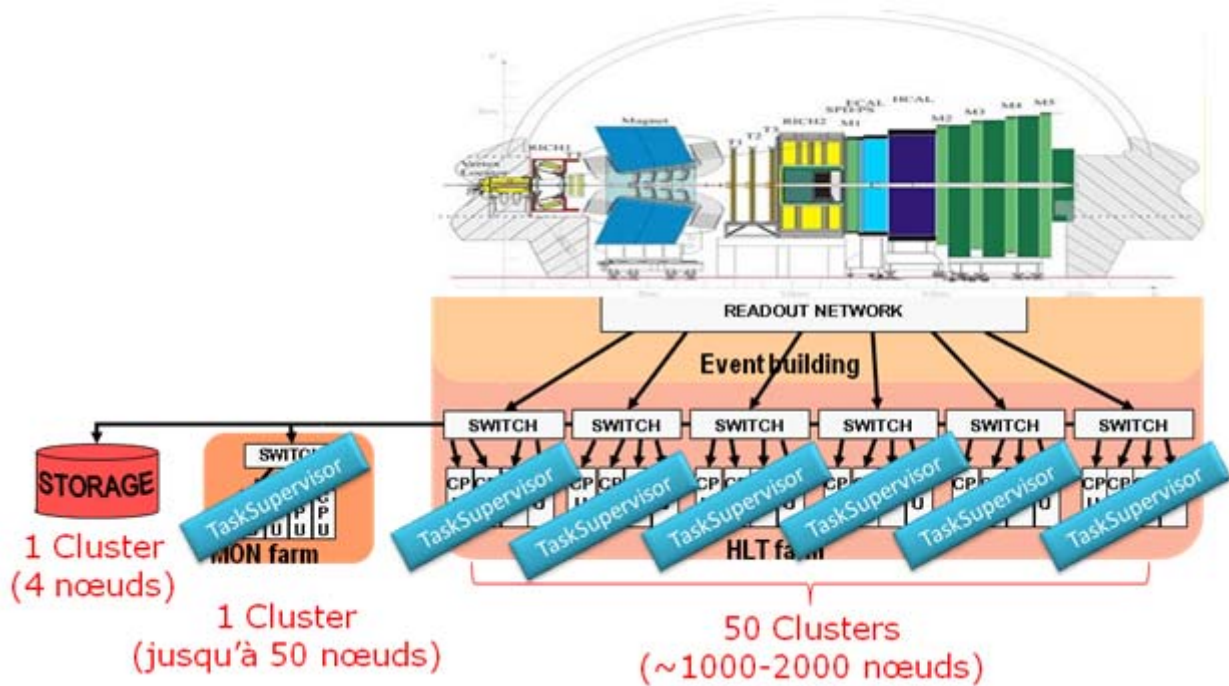


Figure 9: Les TaskSupervisor dans les clusters

Les étapes d'exécution de la solution apportée par le TaskSupervisor peuvent se décomposer en trois :

- Démarrage du TaskSupervisor
- Obtention et traitement de l'information
- Publication d'erreurs

3.2.2 Le diagramme de classes

Les classes utilisées pour le TaskSupervisor sont :

- DimClient
- TaskList
- TmTask
- NodeMonitor
- TaskSupervisor

Voici le diagramme de classes du TaskSupervisor (Figure 10)

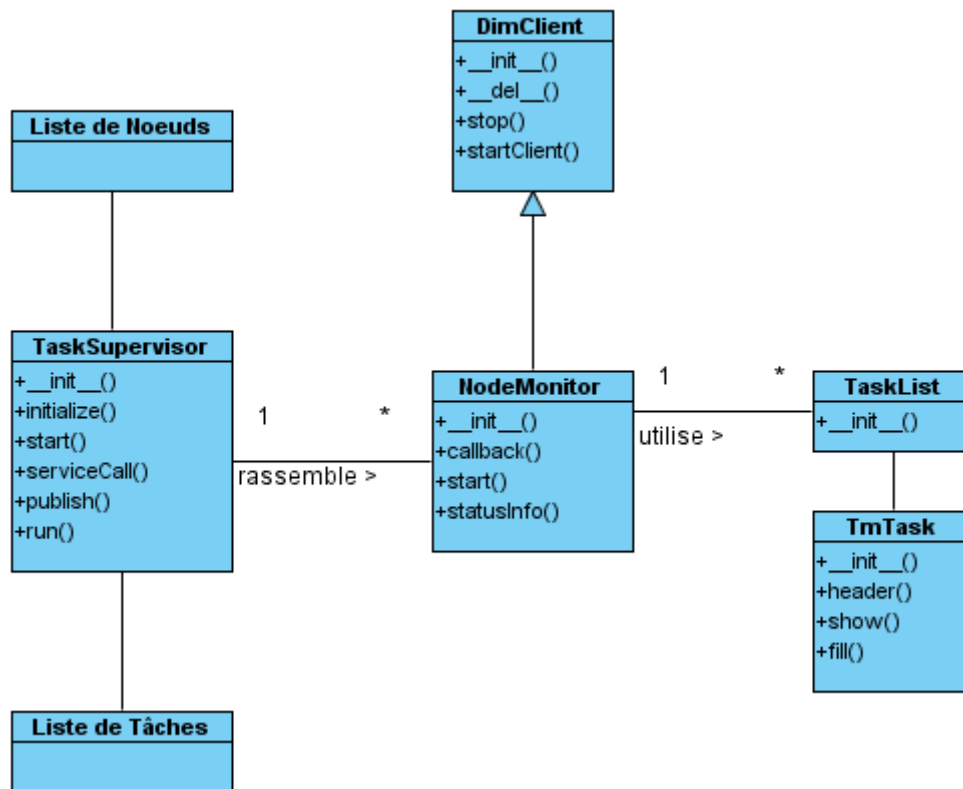


Figure 10: Le diagramme de classes du TaskSupervisor

3.2.3 Démarrage du TaskSupervisor

Pour démarrer le TaskSupervisor, on a besoin de la liste de nœuds du cluster et de la liste de tâches qui lui correspondent. On va appeler un script pour faire l'analyse grammaticale des fichiers et les rendre disponibles dans la mémoire (Figure 11).

```

xml = XMLTaskList.TransformXmlToObjects()
xml.load('../xml/TaskList.xml')
xml.load('../xml/'+os.environ['DIM_DNS_NODE']+'.xml')
    
```

Figure 11: Analyse Grammaticale des fichiers XML pour le TaskSupervisor.

Un défi intéressant a été d'éviter faire une copie différente du TaskSupervisor pour chaque nœud, alors on a pu résoudre le problème avec le paramètre «os.environ['DIM_DNS_NODE']» qui nous a permis de prendre directement du système d'exploitation le nom du nœud où le TaskSupervisor s'exécute.

Une fois que les listes sont dans la mémoire, on démarrera la classe TaskSupervisor() pour commencer leur analyse.

3.2.4 Obtention et Traitement de l'information (NodeMonitor)

Premièrement, avec l'aide de l'analyseur grammatical, il faut réviser si la commande pour verser le contenu des fichiers XML en mémoire a réussi à trouver la liste de nœuds et si la liste avait quelque contenu. Si on n'a pas réussi, dans chaque cas on affichera à l'écran un message d'erreur.

Si tout marche comme il faut, on affichera à l'écran la quantité de nœuds trouvés dans la liste, et finalement on créera une liste avec des commandes individualisées par nœud.

Comme TaskSupervisor est qui doit concentrer l'information de chaque nœud dans le cluster, il fallait aussi programmer une routine qui lui sert comme le démon qui va lui donner l'information de chaque nœud.

Le problème a été résolu en créant la classe NodeMonitor().

NodeMonitor est la routine qui:

- Enquêtera le service « ps/data » du nœud pour avoir accès à ce qui est en train de s'exécuter.
- Fera l'ordonnancement de l'information obtenue de « ps/data ».
- Comparera contre la liste maîtresse de tâches.
- Mettra l'information au format XML et la préparera pour être publiée.

En ayant les commandes pour lancer les NodeMonitor dans une liste de chaîne de caractères, on parcourra la liste pour extraire les éléments et démarrer une instance du NodeMonitor pour chacun. Si on ne peut pas les démarrer, on affiche le message à l'écran. (Figure 12).

```
nodelist = []
for node in self.monitors.keys():
    nodelist.append(node)
    status = self.monitors[node].start()
    if status != S_SUCCESS:
        return error('Failed to start monitor for node:',node)
info('All task monitors for ',self.node,' are now running.')
return S_SUCCESS
```

Figure 12: Lancement des NodeMonitor

Comme le TaskSupervisor doit être un service et NodeMonitor doit demander information à un autre service, alors on a besoin de commencer à utiliser DIM.

En rappelant, DIM est le système de qui permet la communication entre processus et est composé de services qui sont des paquets d'information. Des libraires Python existent déjà pour l'interaction avec le système, alors ça a fait moins compliquée le labeur de programmation, on va utiliser souvent dic qui contient les routines de communication.

Pour le TaskSupervisor j'ai joué avec des différentes façons d'ajouter le service en fonction, finalement les lignes de code dans la Figure 13 montrent la solution adoptée dans le constructeur de la classe. La première ligne a les paramètres à utiliser pour créer le service en invoquant la librairie dic et sa fonction `dis_add_service`, la deuxième ligne est l'instruction pour le démarrer.

TmTask contient les 34 attributs d'une tâche, dans Figure 17, on observe le constructeur de la structure avec les noms des attributs.

Maintenant le problème est la transformation de la chaîne de caractères obtenue de ps/data à la structure créée.

Le cœur du NodeMonitor est la fonction Callback qu'on a programmé pour faire le travail lourd. On a remarqué précédemment la fonction Callback au moment de la souscription au service ps/data.

Sa première obligation est diviser les attributs de la chaîne pour pouvoir remplir la structure TmTask qu'on a créé.

Dans la chaîne de caractères il y a des séparateurs «\», et à chaque occasion qu'on lui trouve pour la 34^{ème} fois on a tous les arguments de la tâche.

Alors j'ai mis une boucle intérieure qui coupe les 34 morceaux pour remplir une structure et une boucle extérieure qui parcourra de 34 en 34 jusqu'à la fin de la chaîne de caractères.

Le boucle extérieure est aussi responsable de créer une nouvelle structure de tâches TmTask au début et l'ajouter à la liste de structures TaskList(list) à la fin.

Le code est dans la Figure 16

```
items = args[0].split('\0')[:-1]
tasks = TaskList()
for i in xrange(len(items)/34):
    t = TmTask()
    t.fill(items[i*34:(i+1)*34])
    tasks.append(t)
node_tasks = self.tasks
```

Figure 16: Boucle principale de NodeMonitor

Maintenant que les structures sont remplies, il fallait obtenir une liste des tâches.

On l'obtient en parcourant la liste de structures et en copiant le UTGID de chaque structure dans une nouvelle liste qu'on utilisera pour comparer contra la liste de maîtresse de tâches qu'on a déjà depuis le démarrage du TaskSupervisor et a été passée comme paramètre au NodeMonitor.

Dans ce point-là on va comparer les deux listes et à chaque fois qu'on trouve une tâche des structures qu'on a juste rempli dans la liste maîtresse en va effacer la tâche de la liste maîtresse, le résidu sera garde dans la liste «all_tasks».

À la fin on a deux possibles résultats :

Le premier est si la liste maîtresse est vide, c'est-à-dire que tout est super.

Mais, le deuxième (qui est le résultat qui nous intéresse) est si la liste n'est pas vide et contient encore des tâches qui n'ont pas été trouvées dans le nœud.

Alors il faut publier le problème.

```
def __init__(self):
    self.utgid = None
    self.tid = None
    self.tgid = None
    self.ppid = None
    self.pgid = None
    self.nlwp = None
    self.user = None
    self.group = None
    self.tty = None
    self.cmd = None
    self.sch = None
    self.prio = None
    self.rtprio = None
    self.nice = None
    self.psr = None
    self.stat = None
    self.perc_cpu = None
    self.perc_mem = None
    self.vsize = None
    self.lock = None
    self.rss = None
    self.data = None
    self.stack = None
    self.exe = None
    self.lib = None
    self.share = None
    self.ignored = None
    self.pending = None
    self.catched = None
    self.blocked = None
    self.started = None
    self.elapsed = None
    self.cputime = None
    self.cmdline = None
```

Figure 17: Structure TmTask

3.2.5 Publication d'erreurs

L'objectif de trouver des tâches qui ne s'exécutent pas a été heureusement accompli.

Mais il y a encore deux autres choses à faire:

- 1) Mettre l'information au format XML et
- 2) La publier

Formater en XML sera fait par le NodeMonitor.

On a le résultat de la comparaison entre la liste maîtresse de tâches et la liste de tâches lue du nœud sous la forme du résidu de la liste maîtresse dans la liste «all_tasks»

Pour tout le processus, on va mettre toujours l'information comme une chaîne de caractères dans la variable «err». On utilisera une stratégie qui le semble aux poupées russes en plaçant des nouveaux éléments XML autour de l'information qui le précède

D'abord, on mettra le nom du nœud avec l'attribut «status= ALIVE » et la compte du numéro de tâches pas trouvées dans la liste. Après dans une boucle, on parcourt la liste ajoutant à chaque cycle le nom d'une tâche. Finalement on ferme chacun des éléments XML qu'on avait ouvert. (Figure 18)

```
if len(all_tasks)>0:
    keys = all_tasks.keys()
    err = '        <Node name="'+self.name+'" status="ALIVE" >\n
          <Missing_Tasks count="%d">\n'% (len(keys),)

    for i in keys:
        utgid = i.replace('<Node>',self.name)
        err = err + '        <Task name="'+utgid+'"/>\n'
    err = err + '        </Missing_Tasks>\n'
    err = err + '        <Found_Tasks count="%d"/>\n'% (num_req_tasks,)
    err = err + '    </Node>\n'
    self.errors = err
else:
    self.errors = None
```

Figure 18: Format d'information en XML

Le NodeMonitor a déjà l'information des erreurs et de son nœud respective et le control est retourné à la classe TaskSupervisor.

Donc le labeur du TaskSupervisor est de déterminer la « santé » du cluster en analysant l'information ramassée de tous ses esclaves (les NodeMonitor).

Le TaskSupervisor cherche dans la liste des NodeMonitor qu'il a démarrés et les demande leurs résultats. Si un NodeMonitor ne répond pas, le nœud est libellé comme MORT (DEAD) et si les NodeMonitor sont trouvés alors leur résultat est publié.

Les états de santé du cluster à déterminer par le TaskSupervisor sont simples :

- ALIVE: S'il n'y a pas de problèmes
- MIXED : S'il y a au moins un nœud mort ou un nœud avec des tâches manquantes.

Alors le TaskSupervisor publiera les résultats des NodeMonitor et ajoutera tout simplement au début l'état de santé du nœud et fermera les éléments XML à la fin. Tout sera mis dans une seule chaîne de caractères «err» qui sera donnée comme paramètre à la commande pour sa publication. (Figure 19)

La fréquence de publication du TaskSupervisor est de 1 seconde.

```
err = '    <Cluster name="'+self.node.upper()+'" status="
if sc1=="DEAD":      err=err+sc1
elif sc2=="ALIVE":   err=err+sc2
else:                err=err+sc1
err = err + ">\n" + er + "    </Cluster>"
res = dimc.dis_update_service(self.serviceId, (err,))
info('dis_update_service:'+str(res)+'\n'+err)
```

Figure 19: Publication d'information du TaskSupervisor

3.3 ClusterMonitor

3.3.1 Définition

Le ClusterMonitor est le programme qui sera en charge de ramasser et publier toute l'information de produit par les TaskSupervisor qui sont partout dans le système de surveillance de l'expérience LHCb. VOIR FIGURE

Au contraire du TaskSupervisor qui existe dans chaque nœud du système et a environ 50 instances en train de s'exécuter, le ClusterMonitor n'a qu'une seule instance dans le système de surveillance. La location du ClusterMonitor peut être quelque cluster dans la ferme. (Figura 20)

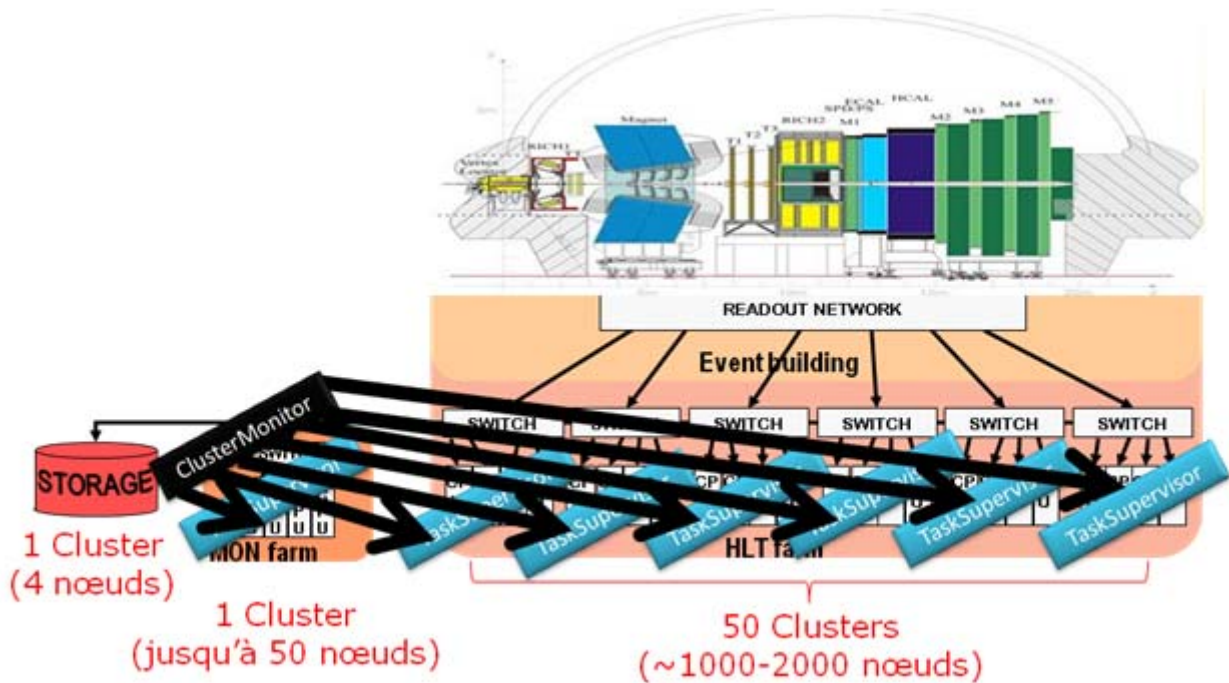


Figura 20 : Le ClusterMonitor dans le système de surveillance

Le ClusterMonitor a 3 étapes principales d'exécution, elles sont :

- Démarrage du ClusterMonitor
- Obtention et traitement de l'information
- Publication d'erreurs

3.3.2 Diagramme de classes

Les classes utilisées pour le ClusterMonitor sont

Classes

- DimClient
- TaskSupervisorClient
- ClusterCollector

Dans la Figure 21 on peut apprécier le diagramme de classes du ClusterMonitor.

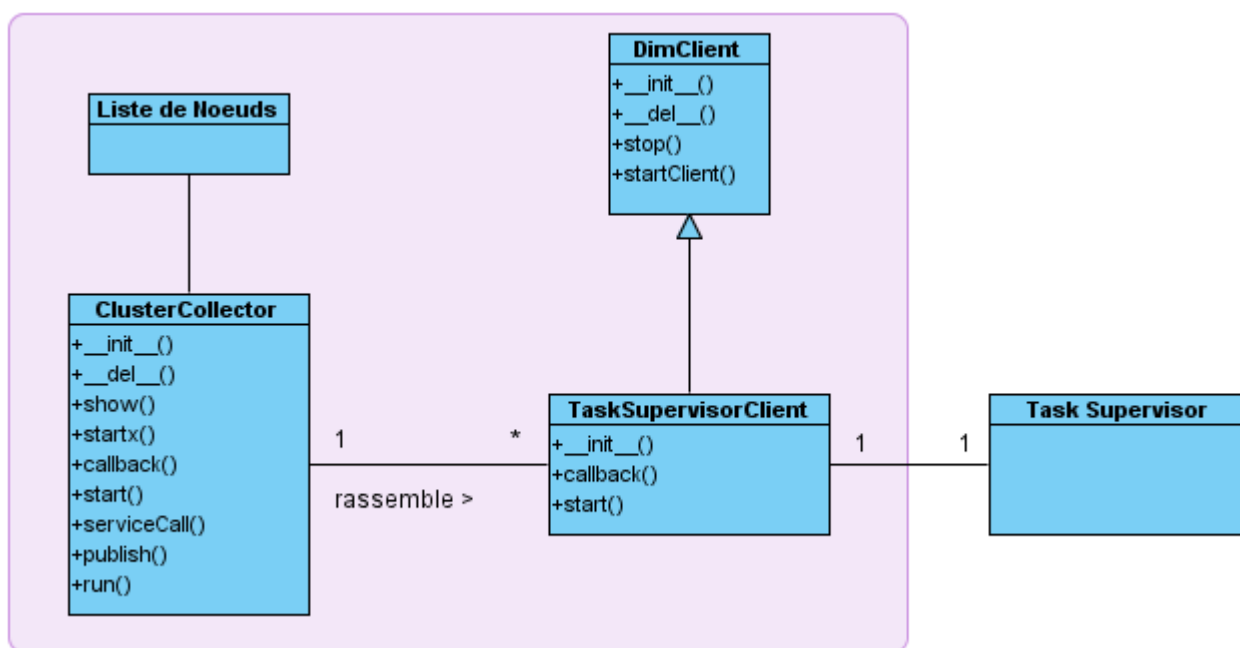


Figure 21: Le diagramme de classes du ClusterMonitor

3.3.3 Démarrage du ClusterMonitor

Le premier dont on a besoin pour commencer à utiliser le ClusterMonitor est de charger les listes de nœuds pour pouvoir localiser les TaskSupervisor. Les listes seront chargées par l'outil d'analyse grammaticale (Figure 22). On aura plusieurs fichiers dont on va extraire le titre de la liste de nœuds pour trouver chacun des TaskSupervisor.

```
xml = XMLTaskList.TransformXmlToObjects()  
xml.load('../xml/mona08.xml')  
xml.load('../xml/hltc08.xml')  
xml.load('../xml/ecs03.xml')
```

Figure 22: Analyse Grammaticale des fichiers XML pour le ClusterMonitor.

En obtenant les noms, on démarrera la classe ClusterCollector.

3.3.4 Obtention et traitement de l'information

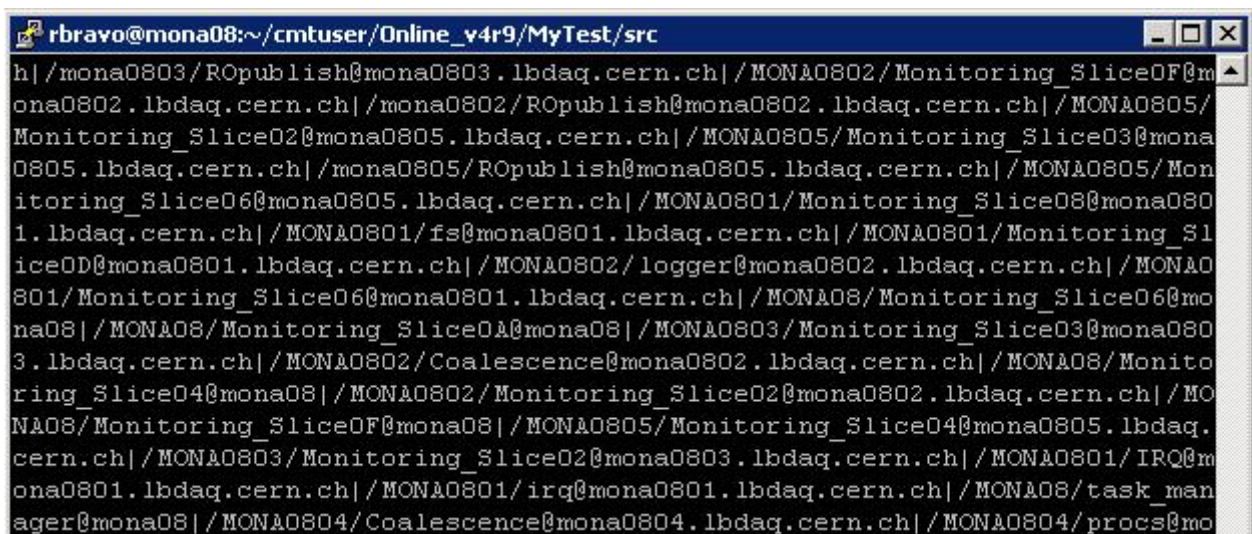
On a déjà une partie de l'information nécessaire pour le ClusterMonitor, pour obtenir le reste il faut se souscrire au service appelé DIS_DNS/SERVER_LIST.

La ligne de code pour se souscrire est Figure 23

```
dimc.dic_info_service('DIS_DNS/SERVER_LIST','C:200000',self.callback)
```

Figure 23: Souscription au service DIS_DNS/SERVER_LIST

Le service DIS_DNS/SERVER_LIST contient la liste de services qui s'exécutent en chaque cluster, de la liste on sera intéressé en obtenir les TaskSupervisor, mais comme d'habitude, il fallait déchiffrer une longue chaîne de caractères. (24)



```
rbravo@mona08:~/cmtuser/Online_v4r9/MyTest/src  
h|/mona0803/ROpublish@mona0803.lbdag.cern.ch|MONA0802/Monitoring_Slice0F@mona0802.lbdag.cern.ch|mona0802/ROpublish@mona0802.lbdag.cern.ch|MONA0805/Monitoring_Slice02@mona0805.lbdag.cern.ch|MONA0805/Monitoring_Slice03@mona0805.lbdag.cern.ch|mona0805/ROpublish@mona0805.lbdag.cern.ch|MONA0805/Monitoring_Slice06@mona0805.lbdag.cern.ch|MONA0801/Monitoring_Slice08@mona0801.lbdag.cern.ch|MONA0801/fs@mona0801.lbdag.cern.ch|MONA0801/Monitoring_Slice0D@mona0801.lbdag.cern.ch|MONA0802/logger@mona0802.lbdag.cern.ch|MONA0801/Monitoring_Slice06@mona0801.lbdag.cern.ch|MONA08/Monitoring_Slice0A@mona08|MONA0803/Monitoring_Slice03@mona0803.lbdag.cern.ch|MONA0802/Coalescence@mona0802.lbdag.cern.ch|MONA08/Monitoring_Slice04@mona08|MONA0802/Monitoring_Slice02@mona0802.lbdag.cern.ch|MONA08/Monitoring_Slice0F@mona08|MONA0805/Monitoring_Slice04@mona0805.lbdag.cern.ch|MONA0803/Monitoring_Slice02@mona0803.lbdag.cern.ch|MONA0801/IRQ@mona0801.lbdag.cern.ch|MONA0801/irq@mona0801.lbdag.cern.ch|MONA08/task_manager@mona08|MONA0804/Coalescence@mona0804.lbdag.cern.ch|MONA0804/procs@mo
```

Figure 24: Chaînes de caractères du service DIS_DNS/SERVER_LIST

Cette fois, on a trois caractères différents qui fonctionnent comme les séparateurs pour notre division de la chaîne de caractères, d'abord « | » qui sert à séparer les données complètes de chaque service, après « @ » qui sépare le nom et le nœud, et finalement « / » pour séparer chaque partie du nom de service

Pour séparer les « | », on travaille avec une boucle où à l'intérieur à chaque fois on divise les « @ » et on garde l'information des deux côtes de l'arobase dans des variables différentes (itm et node)

Une des indications que DIS_DNS/SERVER_LIST nous donne des services est s'ils ont été récemment ajoutés (+) ou supprimés (-), l'indication est optionnel, elle est appliquée seulement aux changements immédiats et peut disparaître avec les mis à jour.

Alors, si on trouve le symbole (+), on assignera une variable de type avec la valeur ADDING; si on trouve le symbole (-), on assigne DYING à la valeur type ; et si on trouve rien, on assigne LISTING.

Ça va nous servir au moment de la dernière division de la chaîne de caractères où on sépare les barres « / » et cherche pour la chaîne 'TaskSupervisor'.

Une fois trouvé l'identifiant du TaskSupervisor on est finalement en position de démarrer les clients qui vont fournir l'information au ClusterCollector.

Si les TaskSupervisor sont accompagnés du type ADDING ou LISTING alors on invoque la classe TaskSupervisorClient() pour chacun et on ajoute le nom du TaskSupervisor à la liste des vivants. Par contre si l'identifiant du TaskSupervisor est marqué avec le type DEAD, on lui ajoute à la liste des services morts.

Le code qui appartient à l'explication donnée pendant les derniers paragraphes est dans la Figure 25.

```
items = args[0].split('|')
for i in items:
    itm,node = i.split('@')
    type = 0
    if itm[0] == '+':
        type = ADDING
        itm = itm[1:]
    elif itm[0] == '-':
        type = DYING
        itm = itm[1:]
    else:
        type = LISTING

    sup = itm[1:].split('/')
    if len(sup)==2 and sup[1]=='TaskSupervisor':
        nam = sup[0].upper()
        if type == ADDING or type == LISTING:
            client = TaskSupervisorClient(nam)
            self.alive[nam] = client
            client.start()
            del self.dead[nam]

        elif type == DYING:
            if self.alive.has_key(nam):
                client = self.alive[nam]
                client.stop()
                self.dead[nam] = client
                del self.alive[nam]
```

Figure 25: Séparation de chaînes dans le ClusterCollector

Comme le TaskSupervisor a son démon NodeClient qui lui fournisse des données, le ClusterMonitor aura son démon TaskSupervisorClient.

Le TaskSupervisorClient reçoit comme paramètre le nom du cluster où le TaskSupervisor se trouve et se souscrit au service pour copier son information.

3.3.5 Publication d'erreurs

Avec la création d'une liste de TaskSupervisor morts et les TaskSupervisorClient souscrit aux TaskSupervisor actifs on a presque tout dont a besoin pour commencer à publier les résultats de tous les TaskSupervisor dans le service ClusterCollector.

Les deux choses qui nous restent, comme dans le TaskSupervisor sont :

- 1) Mettre l'information au format XML et
- 2) La publier

Ici on a un grand avantage pour le premier point, l'information venant des TaskSupervisor est déjà en format XML, alors on a besoin de faire relativement très peu de choses pour préparer l'information.

Finalement dans le ClusterCollector on a crée la fonction show() (Figure 26) pour rassembler les résultats de tous les TaskSupervisor.

D'abord on parcourt la liste des TaskSupervisor morts et on les assigne l'état DEAD dans la au même temps qu'ils sont formatés en XML. En finissant, on parcourt aussi la liste des TaskSupervisor actifs et avec la commande self.alive[n].data on va récupérer les données acquis pour les TaskSupervisorClient.

```
def show(self):
    data = ''
    for n in self.dead.keys():
        data = data + ' <Cluster name="'+n+'" status="DEAD"/>\n'
    for n in self.alive.keys():
        data = data + self.alive[n].data + '\n'
    self.errors = data
    return data
```

Figure 26: La fonction show()

Chapitre 4: Résultats

4.1 Résultats du TaskSupervisor

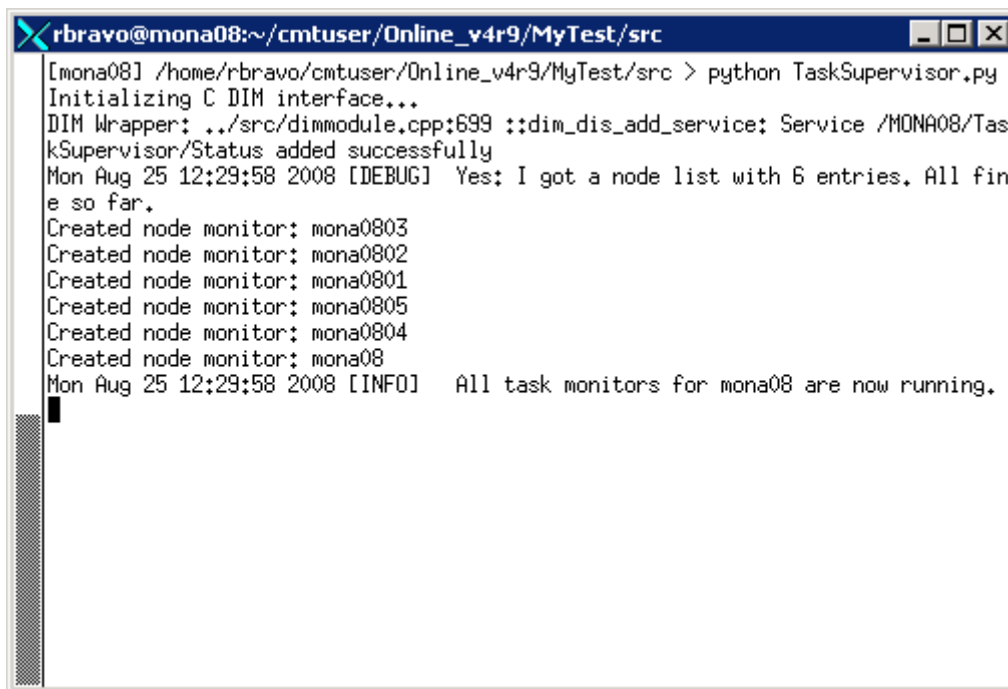
Le TaskSupervisor a été développé pour répondre au besoin de supervision des clusters de traitement d'information et de surveillance de l'expérience LHCb.

Avec le TaskSupervisor on a réussi à savoir quels nœuds d'un cluster ne marchent pas, et pour ceux qui fonctionnent, TaskSupervisor détecte s'il y a des tâches manquants. Tout sert pour déterminer si le nœud fonctionne correctement ou pas.

Finalement, toute l'information est formatée en XML et mise dans un service DIM pour la rendre accessible aux autres processus.

Ce qui suit sont quelques captures d'écran des parties les plus importantes de son fonctionnement.

- Démarrage et création des NodeMonitor (affichage à l'écran) (Figure 27)



```
rbravo@mona08:~/cmtuser/Online_v4r9/MyTest/src > python TaskSupervisor.py
Initializing C DIM interface...
DIM Wrapper: ../src/dimmodule.cpp:699 ::dim_dis_add_service: Service /MONA08/TaskSupervisor/Status added successfully
Mon Aug 25 12:29:58 2008 [DEBUG] Yes: I got a node list with 6 entries. All fine so far.
Created node monitor: mona0803
Created node monitor: mona0802
Created node monitor: mona0801
Created node monitor: mona0805
Created node monitor: mona0804
Created node monitor: mona0808
Mon Aug 25 12:29:58 2008 [INFO] All task monitors for mona08 are now running.
```

Figure 27: Démarrage du TaskSupervisor et création des NodeMonitor

- Contenu du service TaskSupervisor (affichage en DIM) (Figure 28)

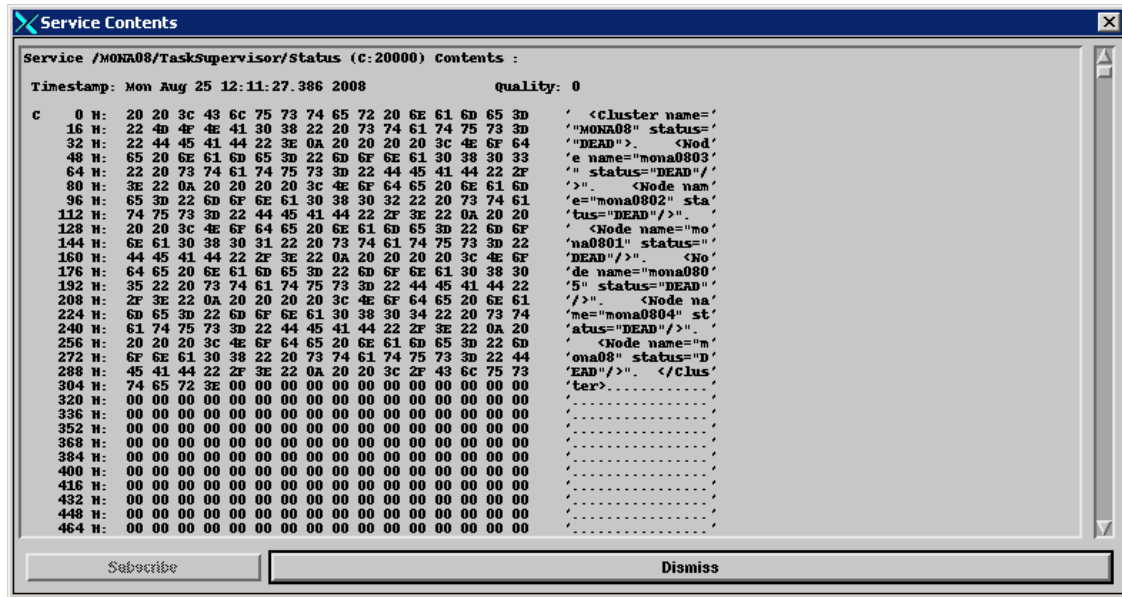


Figure 28: Contenu du service TaskSupervisor (affichage en DIM)

- Contenu du service TaskSupervisor (affichage à l'écran) (Figure 29)

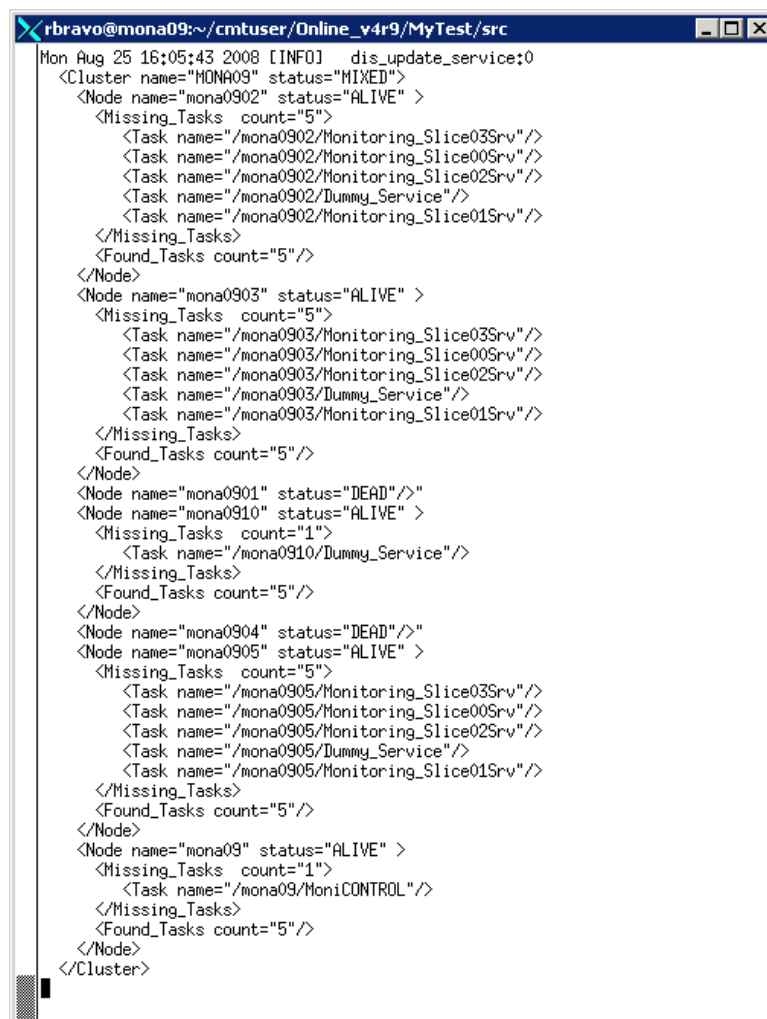


Figure 29: Contenu du service TaskSupervisor (affichage à l'écran)

4.2 Résultats du ClusterMonitor

Les résultats au moment sont encourageants, le ClusterMonitor est l'arrêt obligatoire pour savoir où sont les problèmes parce qu'il est capable de détecter quels clusters ne rend pas information et d'obtenir le contenu des TaskSupervisor de ceux qui fonctionnent.

Jusqu'à maintenant le ClusterMonitor a été testé seulement avec quelques TaskSupervisor, mais pas avec beaucoup au même temps.

Comme dans le TaskSupervisor, le ClusterMonitor fait aussi un format XML pour les données qui s'affichent dans le service ClusterCollector en DIM.

Ce qui suit sont quelques captures d'écran des parties les plus importantes de son fonctionnement.

- Contenu du service ClusterCollector (Figure 30)

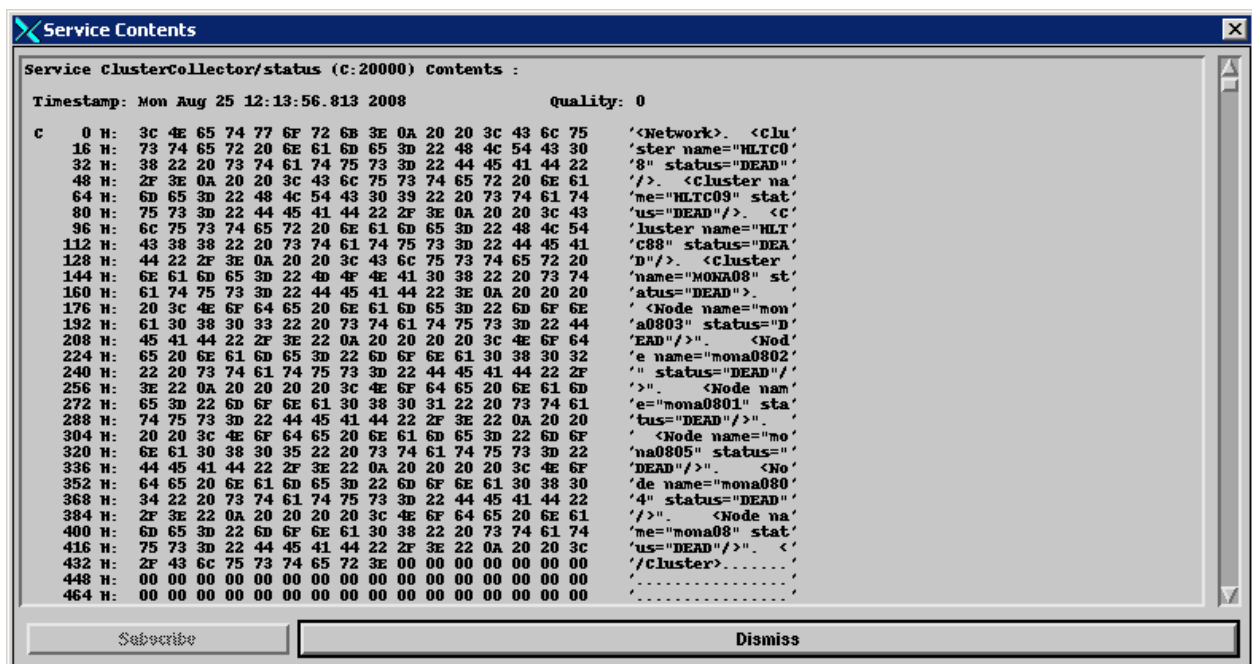
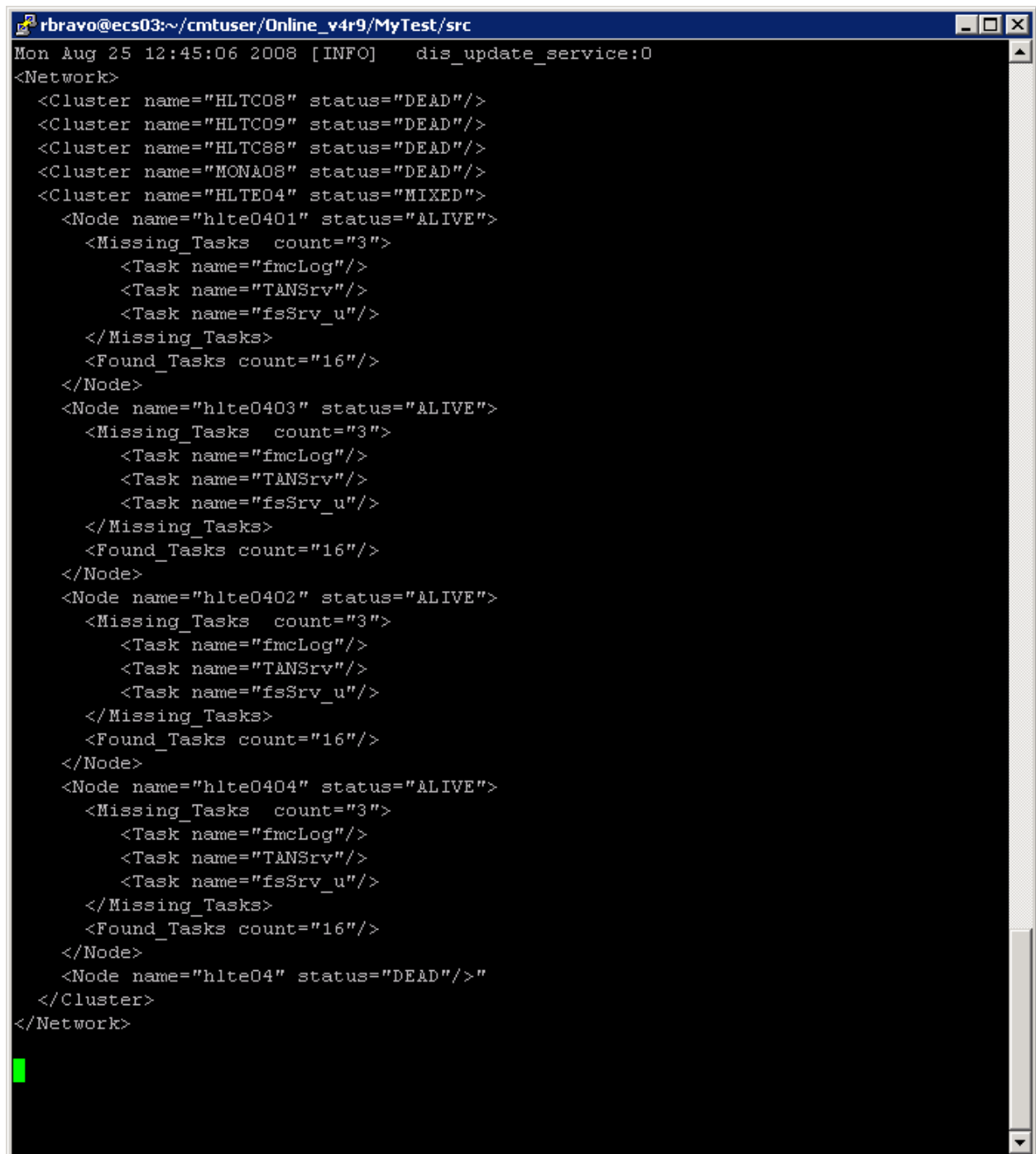


Figure 30: Contenu du service ClusterCollector (affichage en DIM)

- Contenu du service ClusterCollector (affichage à l'écran) (Figure 31)



```

rbravo@ecs03:~/cmtuser/Online_v4r9/MyTest/src
Mon Aug 25 12:45:06 2008 [INFO]    dis_update_service:0
<Network>
  <Cluster name="HLTC08" status="DEAD"/>
  <Cluster name="HLTC09" status="DEAD"/>
  <Cluster name="HLTC88" status="DEAD"/>
  <Cluster name="MONA08" status="DEAD"/>
  <Cluster name="HLTE04" status="MIXED">
    <Node name="hlte0401" status="ALIVE">
      <Missing_Tasks count="3">
        <Task name="fmcLog"/>
        <Task name="TANSrv"/>
        <Task name="fsSrv_u"/>
      </Missing_Tasks>
      <Found_Tasks count="16"/>
    </Node>
    <Node name="hlte0403" status="ALIVE">
      <Missing_Tasks count="3">
        <Task name="fmcLog"/>
        <Task name="TANSrv"/>
        <Task name="fsSrv_u"/>
      </Missing_Tasks>
      <Found_Tasks count="16"/>
    </Node>
    <Node name="hlte0402" status="ALIVE">
      <Missing_Tasks count="3">
        <Task name="fmcLog"/>
        <Task name="TANSrv"/>
        <Task name="fsSrv_u"/>
      </Missing_Tasks>
      <Found_Tasks count="16"/>
    </Node>
    <Node name="hlte0404" status="ALIVE">
      <Missing_Tasks count="3">
        <Task name="fmcLog"/>
        <Task name="TANSrv"/>
        <Task name="fsSrv_u"/>
      </Missing_Tasks>
      <Found_Tasks count="16"/>
    </Node>
    <Node name="hlte04" status="DEAD"/>
  </Cluster>
</Network>

```

Figure 31: Contenu du service ClusterCollector (affichage à l'écran)

- Le ClusterMonitor qui affiche à droite l'information qui obtient du TaskSupervisor à gauche. (Figure 32)

```

rbravo@mona09:~/cmtuser/Online_v4r9/MyTest/src
Tue Aug 26 09:18:54 2008 [INFO] dis_update_service:1
<Cluster name="MONA09" status="MIXED">
  <Node name="mona0902" status="ALIVE" >
    <Missing_Tasks count="5">
      <Task name="/mona0902/Monitoring_Slice03Srv"/>
      <Task name="/mona0902/Monitoring_Slice00Srv"/>
      <Task name="/mona0902/Monitoring_Slice02Srv"/>
      <Task name="/mona0902/Dummy_Service"/>
      <Task name="/mona0902/Monitoring_Slice01Srv"/>
    </Missing_Tasks>
    <Found_Tasks count="5"/>
  </Node>
  <Node name="mona0903" status="ALIVE" >
    <Missing_Tasks count="5">
      <Task name="/mona0903/Monitoring_Slice03Srv"/>
      <Task name="/mona0903/Monitoring_Slice00Srv"/>
      <Task name="/mona0903/Monitoring_Slice02Srv"/>
      <Task name="/mona0903/Dummy_Service"/>
      <Task name="/mona0903/Monitoring_Slice01Srv"/>
    </Missing_Tasks>
    <Found_Tasks count="5"/>
  </Node>
  <Node name="mona0901" status="DEAD"/>
  <Node name="mona0910" status="ALIVE" >
    <Missing_Tasks count="1">
      <Task name="/mona0910/Dummy_Service"/>
    </Missing_Tasks>
    <Found_Tasks count="5"/>
  </Node>
  <Node name="mona0904" status="DEAD"/>
  <Node name="mona0905" status="ALIVE" >
    <Missing_Tasks count="5">
      <Task name="/mona0905/Monitoring_Slice03Srv"/>
      <Task name="/mona0905/Monitoring_Slice00Srv"/>
      <Task name="/mona0905/Monitoring_Slice02Srv"/>
      <Task name="/mona0905/Dummy_Service"/>
      <Task name="/mona0905/Monitoring_Slice01Srv"/>
    </Missing_Tasks>
    <Found_Tasks count="5"/>
  </Node>
  <Node name="mona09" status="ALIVE" >
    <Missing_Tasks count="1">
      <Task name="/mona09/MoniCONTROL"/>
    </Missing_Tasks>
    <Found_Tasks count="5"/>
  </Node>
</Cluster>

rbravo@mona09:~/cmtuser/Online_v4r9/MyTest/src
Tue Aug 26 09:18:54 2008 [INFO] dis_update_service:0
<Network>
  <Cluster name="EC803" status="DEAD"/>
  <Cluster name="HLTC08" status="DEAD"/>
  <Cluster name="HLTC09" status="DEAD"/>
  <Cluster name="HLTC88" status="DEAD"/>
  <Cluster name="MONA08" status="DEAD"/>
  <Cluster name="MONA09" status="MIXED">
    <Node name="mona0902" status="ALIVE" >
      <Missing_Tasks count="5">
        <Task name="/mona0902/Monitoring_Slice03Srv"/>
        <Task name="/mona0902/Monitoring_Slice00Srv"/>
        <Task name="/mona0902/Monitoring_Slice02Srv"/>
        <Task name="/mona0902/Dummy_Service"/>
        <Task name="/mona0902/Monitoring_Slice01Srv"/>
      </Missing_Tasks>
      <Found_Tasks count="5"/>
    </Node>
    <Node name="mona0903" status="ALIVE" >
      <Missing_Tasks count="5">
        <Task name="/mona0903/Monitoring_Slice03Srv"/>
        <Task name="/mona0903/Monitoring_Slice00Srv"/>
        <Task name="/mona0903/Monitoring_Slice02Srv"/>
        <Task name="/mona0903/Dummy_Service"/>
        <Task name="/mona0903/Monitoring_Slice01Srv"/>
      </Missing_Tasks>
      <Found_Tasks count="5"/>
    </Node>
    <Node name="mona0901" status="DEAD"/>
    <Node name="mona0910" status="ALIVE" >
      <Missing_Tasks count="1">
        <Task name="/mona0910/Dummy_Service"/>
      </Missing_Tasks>
      <Found_Tasks count="5"/>
    </Node>
    <Node name="mona0904" status="DEAD"/>
    <Node name="mona0905" status="ALIVE" >
      <Missing_Tasks count="5">
        <Task name="/mona0905/Monitoring_Slice03Srv"/>
        <Task name="/mona0905/Monitoring_Slice00Srv"/>
        <Task name="/mona0905/Monitoring_Slice02Srv"/>
        <Task name="/mona0905/Dummy_Service"/>
        <Task name="/mona0905/Monitoring_Slice01Srv"/>
      </Missing_Tasks>
      <Found_Tasks count="5"/>
    </Node>
    <Node name="mona09" status="ALIVE" >
      <Missing_Tasks count="1">
        <Task name="/mona09/MoniCONTROL"/>
      </Missing_Tasks>
      <Found_Tasks count="5"/>
    </Node>
  </Cluster>
</Network>

```

Figure 32: TaskSupervisor et ClusterCollector au même temps

Les fonctionnes du TaskSupervisor pourront être élargies au future avec l'information qui se stocke de chaque tâche dans la structure de tâches créée. Le ClusterMonitor doit encore être testé avec plus de TaskSupervisor.

À la fin, j'ai laisse à mon superviseur un package logiciel qui contient :

Scripts Python:

- TaskSupervisor.py
- ClusterMonitor.py
- XMLTaskList.py (l'analyseur grammatical utilisé, développé par ma collègue Chourouk)

Fichiers XML:

- TaskList.xml
- Plusieurs fichiers de listes de nœuds comme par exemple mona08.xml

Documentation :

- Fichiers texte avec une brève description des fonctionnes du TaskSupervisor et ClusterMonitor basés dans les commentaires des programmes

Conclusion

Les objectifs principaux du TaskSupervisor, comme trouver des tâches qui ne s'exécutent pas, déterminer l'état de santé d'un nœud et publier les résultats, ont été bien accomplis.

Le TaskSupervisor est encore un peu basic, mais l'un des buts du projet était de créer un outil qui puisse être réutilisable pour des futures applications et améliorations. Un point très important du TaskSupervisor est la structure de tâches où on a gardé tous les arguments de chaque tâche et qui va permettre dans le future de faire des mises à jour du programme en incluant par exemple des routines pour établir des indicateurs de la consommation de ressources tel que la quantité de mémoire ou de CPU utilisée par une tâche, les possibilités sont énormes.

Le ClusterMonitor est un logiciel moins complexe que le TaskSupervisor et une grande partie de son fonctionnement est similaire. Il a été testé dans des conditions de travaux réduit en raison de la quantité de TaskSupervisor qu'on a pu tester en même temps.

Jusqu'à la finalisation de ce rapport, le ClusterMonitor a été testé dans un environnement très réduit (deux ou trois nœuds), mais a fonctionné très bien et on était pour commencer les essais massifs dans la plupart des clusters allaient commencer.

Comme les TaskSupervisor, on a accompli les objectifs principaux du ClusterMonitor comme le rassemblement de données du TaskSupervisor dans un seul lieu, la vérification de présence des nœuds et la publication de résultats.

Aujourd'hui les deux programmes sont encore lancés par ligne de commande, mais prochainement ils seront compris dans l'outil de démarrage FMC pour être exécutés automatiquement.

Au niveau personnel, une des principales difficultés rencontrées a été la façon de manipuler l'information. Dans mon travail antérieur au Mexique, j'avais toujours besoin de mettre l'information dans des fichiers et travailler dans une autre période avec eux pour obtenir les résultats désirés. Mais ici, on a rien écrit dans aucun fichier en temps d'exécution et les résultats devaient être obtenus toujours en travaillant avec ce qu'on avait en mémoire en temps réel.

Références bibliographiques

- [1] GASPAR C., Distributed information Management System, Genève, Suisse, 2002
- [2] CALLOT O. et coll, Online Data Monitoring in the LHCb Experiment, Genève, Suisse, 2005
- [3] FRANK M. et coll, The LHCb High Level Trigger Infrastructure, Genève, Suisse, 2005
- [4] NEUFELD N. LHCb Online Processing and filtering, Victoria, Canada, 2007



Institut **S**upérieur d'Informatique
de **M**odélisation et de leurs **A**pplications

Complexe des Cézeaux
BP 125 – 63173 Aubière Cedex, FRANCE



Organisation **E**uropéenne
pour la **R**echerche **N**ucléaire

CERN
CH-1211 Genève 23, SUISSE

Rapport de stage de 2^{ème} année
Filière 3

Annexes

Présenté par: **Raúl DURÁN BRAVO**
Responsable CERN: **M. Markus FRANK**
Responsable ISIMA : **M. Emmanuel MESNARD**

Du 14 avril au
12 septembre
2008

Annexe A: Code source du TaskSupervisor

"""

Task Supervisor
by: Raul Duran Bravo

It gathers the errors of each Node in a Cluster,
then the summary is published by a service named
/XXXXXX/TaskSupervisor,
where XXXXXX is the name of the Cluster.

The status for each Node can be DEAD and ALIVE.

A Node is declared DEAD if there is no response from it.

A Node is declared ALIVE if it responds, anyway if one
or more Tasks are missing, their names will be specified
and tagged one by one.

Functions that are similar to those in the ClusterMonitor
script are:

- error
- info
- debug
- clearscreen

Classes that are similar to those in the ClusterMonitor
script are:

- DimClient

Classes that are exclusive of TaskSupervisor are:

- TaskList
- TmTask
- NodeMonitor
- TaskSupervisor

Note: It uses an external script named XMLTaskList to

decode the XML information

```
"""
```

```
import os, sys, time
from debug import SAY, ERROR, DEBUG
import dimc, debug
import XMLTaskList
from time import sleep
from threading import Thread, Event
import copy
```

```
S_ERROR = 0
S_SUCCESS = 1
```

```
ADDING = 1
DYING = 2
LISTING = 3
```

```
def error(*args):
    """
        The error function labels the information as
        [ERROR] to be properly identified when showed
    """
    s = ''
    for i in args:
        s = s + str(i)
    print '%s %-8s %s'%(time.ctime(), '[ERROR]', s)
    return S_ERROR
```

```
def info(*args):
    """
        The info function labels the information as
        [INFO] to be properly identified when showed
    """
    s = ''
    for i in args:
        s = s + str(i)
    print '%s %-8s %s'%(time.ctime(), '[INFO]', s)
```

```

def debug(*args):
    """
        The debug function labels the information as an
        [DEBUG] to be properly identified when showed
    """
    s = ''
    for i in args:
        s = s + str(i)
    print '%s %-8s %s'%(time.ctime(), '[DEBUG]', s)

def clearscreen(numlines=100):
    """
        The clearscreen function is invoked each second
        to sweep the old information from the screen and
        permits the updates to be posted without displacing
        the info from its regular place, it also takes into
        account the type of operating system to do it.
    """
    if os.name == "posix":
        # Unix/Linux/MacOS/BSD/etc
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        # DOS/Windows
        os.system('CLS')
    else:
        # Fallback for other operating systems.
        print '\n' * numlines

class DimClient:
    """
        The DimClient class sets the constructor and the
        destructor for a future DIM Service as well as
        the cleanup after stopping a service.

        Also, the base parameters for starting the service
        are declared here.
    """

```

```

    TaskSupervisor and ClusterSupervisor inherit from
    this class
"""
def __init__(self,name):
    "Standard constructor"
    self.name = name
    self.id = None
def __del__(self):
    if self.id is not None:
        print 'Release service:',self.id
        dimc.dic_release_service(self.id)
def stop(self):
    if self.id is not None:
        dimc.dic_release_service(self.id)
    self.id = None
    return self

def startClient(self,datapoint,format,callback):
    if self.id is not None:
        dimc.dic_release_service(self.id)
    self.id = dimc.dic_info_service(datapoint,format,callback)
    return S_SUCCESS

class TaskList(list):
    """
        An instance of the TaskList class represents a collection of
        task object as it results from reading the XML database.
    """
    def __init__(self):
        "Standard constructor"
        list.__init__(self)

class TmTask:
    """
        # initialization of the structure with 34 attributes per task

        The TmTask class initializes the structure of the 'Task'
        object with the 34 attributes of each task and when its

```

fill function is invoked it assigns a number to each one of the attributes to be filled.

HEADER and SHOW are samples of the information that can be displayed.

```
"""
def __init__(self):
    self.utgid = None
    self.tid = None
    self.tgid = None
    self.ppid = None
    self.pgid = None
    self.nlwp = None
    self.user = None
    self.group = None
    self.tty = None
    self.cmd = None
    self.sch = None
    self.prio = None
    self.rtprio = None
    self.nice = None
    self.psr = None
    self.stat = None
    self.perc_cpu = None
    self.perc_mem = None
    self.vsize = None
    self.lock = None
    self.rss = None
    self.data = None
    self.stack = None
    self.exe = None
    self.lib = None
    self.share = None
    self.ignored = None
    self.pending = None
    self.caught = None
    self.blocked = None
    self.started = None
```

```

self.elapsed = None
self.cputime = None
self.cmdline = None

def header(self): # header for printing some of the values (use with show()) currently unused
    print '%-32s %6s %6s %6s'%( 'UTGID', 'TID', 'TGID', 'VSIZE' )

def show(self):    # gathering some values to print (currently unused)
    print '%-32s %6d %6d %6d'%(self.utgid,self.tid,self.tgid,self.vsize)

def fill(self,items):    # filling the attribute structure
    self.utgid = items[0]
    self.tid = int(items[1])
    self.tgid = int(items[2])
    self.ppid = int(items[3])
    self.pgid = items[4]
    self.nlwp = items[5]
    self.user = items[6]
    self.group = items[7]
    self.tty = items[8]
    self.cmd = items[9]
    self.sch = items[10]
    self.prio = items[11]
    self.rtprio = items[12]
    self.nice = items[13]
    self.psr = items[14]
    self.stat = items[15]
    self.perc_cpu = items[16]
    self.perc_mem = items[17]
    self.vsize = int(items[18])
    self.lock = items[19]
    self.rss = items[20]
    self.data = items[21]
    self.stack = items[22]
    self.exe = items[23]
    self.lib = items[24]
    self.share = items[25]
    self.ignored = items[26]
    self.pending = items[27]

```

```

self.catched = items[28]
self.blocked = items[29]
self.started = items[30]
self.elapsed = items[31]
self.cputime = items[32]
self.cmdline = items[33]

```

```

class NodeMonitor(DimClient):

```

```

    """

```

An instance of the NodeMonitor class polls the /ps/data service in the current Node to see what tasks are being executed in that moment.

The callback function defined in the class splits the arguments read from /ps/data into 34 pieces which is the number of attributes each task has. A TmTask structure is then filled with all the 34 attributes for each existing task.

The utgid names are then put together in a list that is compared against the Master Tasklist. The Master Tasklist is provided with anticipation, right now they are static XML files.

If at the end of the comparison, there is a difference between the MasterList and the created utgid list, the tasks from the Masterlist that were not found in the utgid list will be formatted as XML information for later publishing

The NodeMonitor class inherits from the DimClient.

```

    """

```

```

def __init__(self,node,tasks):
    "Standard constructor"
    DimClient.__init__(self,node)
    self.tasks = tasks

```

```

self.errors = None
self.time = 0
print 'Created node monitor:',self.name

def callback(self,*args): # separates each item from the chain
    try:
        if len(args)==0:\
            return
        self.time = time.time()
        items = args[0].split('\0')[:-1] # the long chain of arguments is splitted here
        tasks = TaskList()

        for i in xrange(len(items)/34):
            t = TmTask()
            # calling of the fill routine that will put the arguments into the object
            t.fill(items[i*34:(i+1)*34])
            tasks.append(t)
        node_tasks = self.tasks
        all_tasks = {}

        for l in node_tasks.taskLists.keys():
            for t in node_tasks.taskLists[l].tasks.keys():
                all_tasks[t] = node_tasks.taskLists[l].tasks[t]
        num_req_tasks = len(all_tasks.keys())

        for t in tasks:
            utgid = t.utgid.replace(self.name, '<Node>')
            if all_tasks.has_key(utgid):
                del all_tasks[utgid]

        if len(all_tasks)>0:
            keys = all_tasks.keys()
            err = '      <Node name="'+self.name+'" status="ALIVE" >\n      <Missing_Tasks
count="%d">\n'%(len(keys),)
            for i in keys:
                utgid = i.replace('<Node>',self.name)
                err = err + '      <Task name="'+utgid+'"/>\n'
            err = err + '      </Missing_Tasks>\n'
            err = err + '      <Found_Tasks count="%d"/>\n'%(num_req_tasks,)

```

```

        err = err + '    </Node>\n'
        self.errors = err
    else:
        self.errors = None

except Exception,X:
    import traceback
    traceback.print_stack()
    print 'Something went wrong:',str(X)
    print 'Exception handling finished.'

def start(self):
    #info('Node monitor for ',self.name,' started.....')
    return self.startClient('/'+self.name.upper()+'/ps/data','C:200000',self.callback)

def statusInfo(self):
    if (time.time() - self.time) > 35:
        #print 'No answer since ', self.time, time.time() - self.time
        self.errors = '    <Node name="'+self.name+" status="DEAD"/>"\n'
    return self.errors

```

```

class TaskSupervisor:

```

```

    """

```

An instance of the TaskSupervisor class is able to start the NodeMonitors in each node and then retrieve their informations to publish it in each node in a service called after it: TaskSupervisor.

First, the constructor initializes the necessary values, one of them is the name of the service (with the node name included).

The initialize function using the XMLTaskList script checks the existence of nodes. Then the start function makes each NodeMonitor begin their work.

The run function invokes the publish function each second. Then the publish function polls for the status of the NodeMonitors, based on their response, it assigns DEAD, MIXED or ALIVE status and publishes.

The results are framed in XML tags to match the information received from the NodeMonitors

An instance of this class is able to supervise a collection of processing nodes. To do so it will:

- Query the task inventory to retrieve the list of tasks executing on the controls PC and the worker nodes.
- Delegate the checking of the task existence to the node class and process the return code indicating if a node works properly or not.
- Publish the result indicating if ALL nodes work properly or show errors.

@author R.Bravo

@version 1.0

"""

```
def __init__(self,name,inventory):
    "Iniitalize the object"
    self.node = name
    self.name = '/' + name.upper() + '/TaskSupervisor'
    self.inventory = inventory
    self.nodes = None
    self.monitors = {}
    svc = self.name + '/Status'
    #dimc.dis_set_dns_node('ecs03') # a call can be individually made with this command
    dimc.dis_set_dns_node(os.environ['DIM_DNS_NODE'])

    self.serviceId = dimc.dis_add_service(svc, 'C:20000', self.serviceCall, 1)
    dimc.dis_start_serving(self.name)
    dimc.dic_set_dns_node(name)
    self.list = {}
```

```

def initialize(self):
    "Load the database and check for consistency"
    nodes = self.inventory.getNodes()
    if nodes is None:
        return error('Failed to retrieve the nodelist from the inventory.')
    elif len(nodes) == 0:
        return error('Node list for ',self.node,' is of ZERO length. This cannot be.')
    else:
        debug('Yes: I got a node list with ',len(nodes),' entries. All fine so far.')
    self.nodes = nodes
    for node in self.nodes.keys():
        self.monitors[node] = NodeMonitor(node,self.nodes[node])
    return S_SUCCESS

def start(self):
    "Start controlling the associated nodes."
    nodelist = []
    for node in self.monitors.keys():
        nodelist.append(node)
        status = self.monitors[node].start()
        if status != S_SUCCESS:
            return error('Failed to start monitor for node:',node)
    info('All task monitors for ',self.node,' are now running.')
    #print nodelist
    return S_SUCCESS

def serviceCall(self,*args):
    print 'serviceCall-----'
    print 'serviceCall',args
    return ('whatever',)

def publish(self):
    er = ''
    sc1 = "DEAD"
    sc2 = "ALIVE"
    clearscreen()
    for node,mon in self.monitors.items():
        e = mon.statusInfo()
        if e is not None:

```

```

        dead = e.find('status="DEAD">0
        alive = e.find('status="ALIVE">0
        if alive and sc1=="DEAD":
            sc1 = "MIXED"
        if dead and sc2=="ALIVE":
            sc2 = "MIXED"
        er = er + e
    err = ' <Cluster name="'+self.node.upper()+'" status="'
    if sc1=="DEAD": err=err+sc1
    elif sc2=="ALIVE": err=err+sc2
    else: err=err+sc1
    err = err + '>\n' + er + " </Cluster>"
    res = dimc.dis_update_service(self.serviceId,(err,))
    info('dis_update_service:'+str(res)+'\n'+err)

def run(self):
    "Here I can sleep while my monitoring slaves actually have to struggle."
    collect=0
    while(1):
        collect = collect + 1
        time.sleep(1)
        if (collect%1)==0:
            self.publish()

def callTaskSupervisor():
    """
        The callTaskSupervisor function loads the Master Tasklist,
        the TaskInventory and the Nodelist. It also starts the
        TaskSupervisor
    """
    xml = XMLTaskList.TransformXmlToObjects()
    xml.load('../xml/TaskList.xml') # loads the Task List
    xml.load('../xml/'+os.environ['DIM_DNS_NODE']+'.xml') # loads the Node List

    sup = TaskSupervisor(os.environ['DIM_DNS_NODE'],xml)
    sup.initialize()

```

```
sup.start()  
sup.run()
```

```
if __name__=="__main__":  
    callTaskSupervisor()
```

Annexe B: Code source du ClusterMonitor

"""

Cluster Monitor
by: Raul Duran Bravo

This script collects the information of the TaskSupervisors that are running in the clusters and then publishes the collective information creating a service called 'ClusterCollector' in the cluster where it was invoked.

Failing status for Clusters are DEAD and MIXED.

A cluster is declared DEAD when its TaskSupervisor could not be interrogated for whatever reason.

A cluster is declared MIXED when its TaskSupervisor exists but has error messages, then it proceeds to publish the information obtained from the corresponding TaskSupervisor.

Functions that are similar to those in the TaskSupervisor script are:

- error
- info
- debug
- clearscreen

Classes that are similar to those in the TaskSupervisor script are:

- DimClient

Classes that are exclusive of ClusterMonitor are:

- TaskSupervisorClient
- ClusterSupervisor
- ClusterCollector

Note: It uses an external script named XMLTaskList to decode the XML information

```
"""

import os, sys, time
from debug import SAY, ERROR, DEBUG
import dimc, debug
import XMLTaskList
from time import sleep
from threading import Thread, Event
import copy

S_ERROR = 0
S_SUCCESS = 1

ADDING = 1
DYING = 2
LISTING = 3

def error(*args):
    """
        The error function labels the information as
        [ERROR] to be properly identified when showed
    """
    s = ''
    for i in args:
        s = s + str(i)
    print '%s %-8s %s'%(time.ctime(), '[ERROR]', s)
    return S_ERROR

def info(*args):
    """
        The info function labels the information as
        [INFO] to be properly identified when showed
    """
```

```

s = ''
for i in args:
    s = s + str(i)
print '%s %-8s %s'%(time.ctime(), '[INFO]', s)

def debug(*args):
    """
        The debug function labels the information as an
        [DEBUG] to be properly identified when showed
    """
    s = ''
    for i in args:
        s = s + str(i)
    print '%s %-8s %s'%(time.ctime(), '[DEBUG]', s)

def clearscreen(numlines=100):
    """
        The clearscreen function is invoked each second
        to sweep the old information from the screen and
        permits the updates to be posted without displacing
        the info from its regular place, it also takes into
        account the type of operating system to do it.
    """
    if os.name == "posix":
        # Unix/Linux/MacOS/BSD/etc
        os.system('clear')
    elif os.name in ("nt", "dos", "ce"):
        # DOS/Windows
        os.system('CLS')
    else:
        # Fallback for other operating systems.
        print '\n' * numlines

class DimClient:
    """
        The DimClient class sets the constructor and the
        destructor for a future DIM Service as well as
    """

```

the cleanup after stopping a service. Also, the base parameters for starting the service are declared here.

TaskSupervisor and ClusterSupervisor inherit from this class

```
"""
def __init__(self,name):
    "Standard constructor"
    self.name = name
    self.id = None
def __del__(self):
    if self.id is not None:
        print 'Release service:',self.id
        dimc.dic_release_service(self.id)
def stop(self):
    if self.id is not None:
        dimc.dic_release_service(self.id)
    self.id = None
    return self

def startClient(self,datapoint,format,callback):
    if self.id is not None:
        dimc.dic_release_service(self.id)
    self.id = dimc.dic_info_service(datapoint,format,callback)
    return S_SUCCESS
```

```
class TaskSupervisorClient(DimClient):
    """
        An instance of the TaskSupervisorClient class subscribes
        to the TaskSupervisor service in a node.
    """
    def __init__(self,name):
        DimClient.__init__(self,name)
        self.data = ''
```

```

def callback(self, *args):
    try:
        if len(args)>0:
            self.data = args[0]
        else:
            info('TaskSupervisorClient received invalid data.')

    except Exception,X:
        import traceback
        traceback.print_stack()
        print 'Something went wrong:',str(X)
        print 'Exception handling finished.'

def start(self):
    #info('TaskSupervisorClient for ',self.name,' started.....')
    return self.startClient('/'+self.name.upper()+'/TaskSupervisor/Status','C:20000',self.callback)

```

```

class ClusterCollector:

```

```

    """

```

An instance of the ClusterCollector class subscribes to the DIS_DNS/SERVER_LIST service and extracts the list of processes that are running in each node of the cluster.

It also looks for added and removed processes to put them in the ALIVE or DEAD lists.

Then the information is formatted in XML and published by a process called ClusterCollector

```

    """

```

```

def __init__(self,inventory,name='ClusterCollector'):
    self.name = name
    self.svcID = None
    self.dead = {}
    self.alive = {}
    self.errors = ''
    self.inventory = inventory

```

```

self.monitors = {}
self.node = name
svc = self.name+'/status'
self.serviceId = dimc.dis_add_service(svc, 'C:20000',self.serviceCall,1)
dimc.dis_start_serving(self.name)
lst = self.inventory.getNodeLists()
s = ''

for i in lst.keys():
    l = lst[i]
    s = s + ' ' + l.Name+ ' ['+str(len(l.nodes))+'] '
    client = TaskSupervisorClient(l.Name)
    self.dead[l.Name.upper()] = client
    #print s

def __del__(self):
    if self.svcID is not None:
        dimc.dic_release_service(self.svcID)

def show(self):
    data = ''

    for n in self.dead.keys():
        data = data + ' <Cluster name="'+n+'" status="DEAD"/>\n'

    for n in self.alive.keys():
        data = data + self.alive[n].data + '\n'
    self.errors = data
    #info('State summary')
    return data

def startx(self):
    """Connect to the DIS_DNS/SERVER_LIST datapoint. Then:
    -- Select all available TaskSupervisor status datapoints status datapoint?

    -- Collect the result
    -- Publish the summary information to be eventually
    put into PVSS.
    """

```

```

self.svcID=dimc.dic_info_service('DIS_DNS/SERVER_LIST', 'C:200000', self.callback)
return self

def callback(self, *args):
    """
    """
    try:
        #print len(self.alive)
        if len(args) == 0:
            print 'There is nothing in DIS_DNS/SERVER_LIST '
            return
        items = args[0].split('|')

        for i in items:
            itm,node = i.split('@')
            type = 0
            if itm[0] == '+':
                type = ADDING
                itm = itm[1:]
            elif itm[0] == '-':
                type = DYING
                itm = itm[1:]
            else:
                type = LISTING

            sup = itm[1:].split('/')
            if len(sup)==2 and sup[1]=='TaskSupervisor':
                nam = sup[0].upper()
                if type == ADDING or type == LISTING:
                    client = TaskSupervisorClient(nam)
                    self.alive[nam] = client
                    client.start()
                    del self.dead[nam]
                    #print 'Added new client:',nam, ' to ALIVE list'
                elif type == DYING:
                    if self.alive.has_key(nam):
                        client = self.alive[nam]
                        client.stop()
                        self.dead[nam] = client

```

```

        del self.alive[nam]
        #print 'Moved client:',nam,' to DEAD list'

except Exception,X:
    print str(X)

def start(self):
    "Start controlling the associated nodes."
    for node in self.monitors.keys():
        status = self.monitors[node].start()
        if status != S_SUCCESS:
            # This should never fail, because nobody is monitoring me!!!!
            return error('Failed to start monitor for node:',node)
    info('All task monitors for ',self.node,' are now running.')
    return S_SUCCESS

def serviceCall(self,*args):
    print 'serviceCall-----'
    print 'serviceCall'
    return ('whatever',)

def publish(self):
    err = "<Network>\n"
    err = err + self.show()
    err = err + "</Network>\n"
    res = dimc.dis_update_service(self.serviceId,(err,))
    # all errors are published here
    info('dis_update_service:'+str(res)+'\n'+err)

def run(self):
    "Here I can sleep while my monitoring slaves actually have to struggle."
    collect=0
    while(1):
        collect = collect + 1
        time.sleep(1)
        if (collect%1)==0:
            clearscren()
            self.publish()

```

```

def clusterMonitor():
    """
        The clusterMonitor function loads the necessary lists,
        for now, the XML Nodelists are static. After that, the
        ClusterCollector is started
    """
    xml = XMLTaskList.TransformXmlToObjects()
    #xml.load('../xml/TaskList.xml')      # loads the Task List
    #xml.load('../xml/TaskInventory.xml') # loads the Task Inventory
    xml.load('../xml/mona08.xml')         # loads the Node List
    xml.load('../xml/hltc08.xml')         # loads the Node List
    xml.load('../xml/ecs03.xml')          # loads the Node List

    collector = ClusterCollector(xml)
    collector.startx()
    collector.run()

if __name__=="__main__":
    clusterMonitor()

```