An autonomic approach to configure HEP (High Energy Physics) experiments, applied to LHCb (Large Hadron Collider beauty)

Une approche « autonomic » pour la configuration d'une expérience PHE (Physique des Hautes Energies), appliquée à LHCb (Large Hadron Collider beauty)

Lana Abadie, October 2006

A thesis submitted for the degree of Doctor of Philosophy in Computing Sciences of the University Paris VI

Thesis supervised by Monique Becker
Defended December 15th , 2006

Jury:

Claude Girault	**: Président du Jury**
Ad Aerts	**: Rapporteur**
Dirk Duellmann	**: Rapporteur**
Philippe Charpentier	**: Examinateur**
Monique Becker	**: Directeur**
Eric van Herwijnen	**: Co-directeur**
Pierre Vincent	**: Co-directeur**

To the most wonderful parents in the world

Impose ta chance, serre ton bonheur, va vers ton risque…
A te regarder, ils s'habitueront.

RENE CHAR, Les Matinaux.

Table of Contents

Table of Figures

Abstract

Properly configuring an HEP (High Energy Physics) experiment becomes a more and more complex task as the number of electronics modules grows and technologies evolve quickly. Anticipating a fault in the software or in the hardware during the configuration or the data taking requires an adaptive and modular control system.

The introduction of autonomic tools and databases in the HEP world is quite recent and contributes to implement a more reliable system. The LHCb control system innovates as it has been built using autonomic tools. The main contribution of this PhD is the implementation of an autonomic 3-Tier architecture to configure the LHCb experiment which is a huge network of devices of different types, and its integration in the control system. This new type of autonomics architecture consists of:

- A database layer. A relational Oracle database implemented using the Oracle technology contains the information required for configuration, namely about configuration parameters, the connectivity of the experiment and the inventory/history of devices. A method based on prime numbers has been introduced to replace a N:M relationship. PL/SQL applications have been built to automate several steps in the configuration.
- An object layer. A set of smart libraries has been written to allow manipulating the information stored in the database in a safe and consistent manner and without any knowledge of the database table schema.
- A GUI layer. Users view and modify the content of the database using user-friendly graphical interfaces.

Each layer has been tested and validated. The behaviour of the whole architecture and its integration in the control system have also been tested successfully.

Keywords: autonomic, configuration, equipment management, graph, networking, prime numbers, relational database, SCADA system.

Résumé

Configurer une expérience PHE (Physique des Hautes Energies) devient de plus en plus complexe : le nombre de modules augmente et de nouvelles technologies sont utilisées. Anticiper les fautes d'un software ou d'un hardware nécessite un système de contrôle robuste.

Les systèmes autonomes et dynamiques ont été introduits récemment dans le monde PHE afin d'améliorer la robustesse des expériences. L'architecture du système de contrôle de LHCb qui intègre des outils *autono*mes et dyna*miques* (*autonomique*) constitue une innovation. La contribution de cette thèse est l'implémentation d'une architecture 3 tiers « autonomic » permettant de configurer l'expérience LHCb, i.e. un gigantesque réseau d'équipement. Ce nouveau type d'architecture « autonomic » comprend :

- une couche accès aux données. Une base de données relationnelle Oracle contient des informations sur les paramètres de configuration, sur la topologie de l'expérience et l'inventaire des modules. Une méthode utilisant les nombres premiers est présentée pour remplacer une relation N:M. Des applications PL/SQL ont été implémentées pour automatiser certaines étapes de la configuration.
- une couche traitement des données. Des librairies de fonctions intelligentes permettent de manipuler les informations contenues dans la base de données de manière sûre et cohérente et sans connaissance préalable de la structure de la base de données.
- une couche présentation des données. Des interfaces graphiques permettent de modifier et de visualiser les informations stockées.

Chaque couche a été testée et validée. Le fonctionnement de l'architecture globale et son intégration dans le système de contrôle ont été validés avec succès.

Mots clés : autonomique, bases de données, configuration de réseaux, gestion d'équipement, graphe, PHE, modèle relationnel, nombres premiers, tables de routage, topologie des systèmes.

Résumé par chapitre de la thèse (en français)

Chapitre 1 Introduction à l'expérience LHCb

Les expériences PHE étudient les particules élémentaires (les photons, électrons, etc.) et leurs interactions. Elles tentent de répondre à des questions fondamentales telles que l'origine de la masse, l'origine de l'Univers.

L'expérience LHCb a pour but d'étudier la violation CP (charge parité) sur les particules B-mésons. Le détecteur LHCb (voir Figure 1) est composé de plusieurs sous détecteurs (VELO, PUS, RICH, Silicon Trackers, Outer Trackers, Calorimètres, et un détecteur à muons). Un gigantesque aimant permet de faire courber la trajectoire des particules.

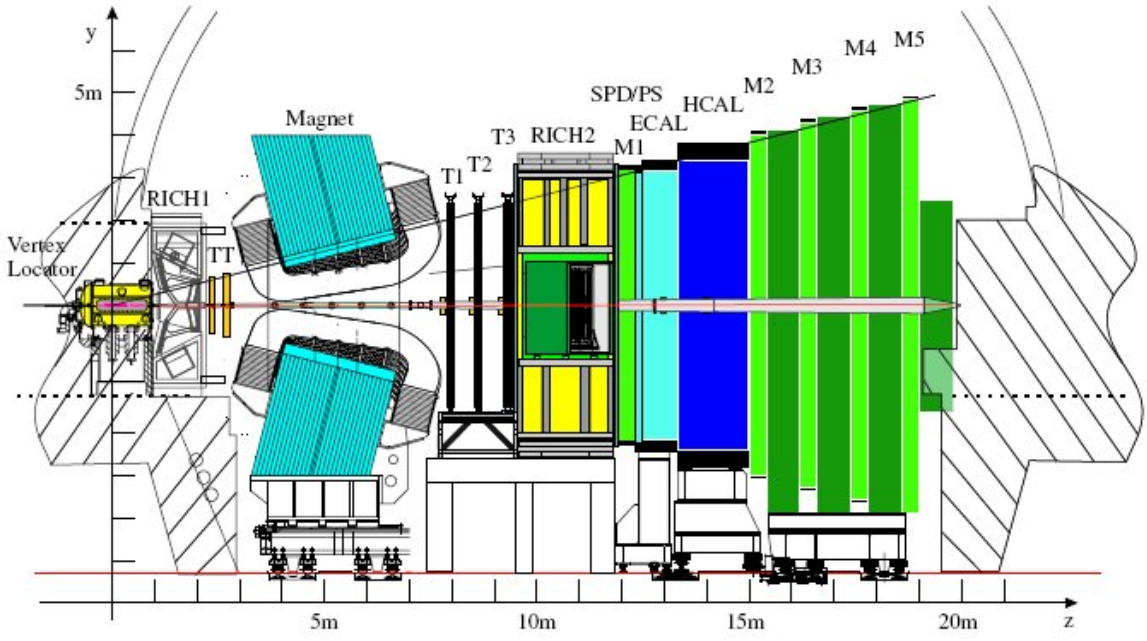


Figure 1. Vue latérale du détecteur LHCb.

Outre le détecteur, l'expérience comprend un système dit « Online », composé de quatre éléments :

- Le Trigger. Les collisions ont lieu toutes les 25ns. Elles sont détectées par des capteurs situés sur le détecteur et produisent des signaux arrivant donc à une fréquence de 40 MHz. Ce qui représente une importante quantité de données! Il est donc nécessaire de filtrer ces signaux. C'est le travail du Trigger, qui dans le cas du détecteur LHCb comprend deux niveaux. Le premier niveau (L0 Trigger) s'appuie sur du hardware. Les données sont ainsi réduites à la fréquence de 1 MHz. Le deuxième niveau (HLT) est de nature logicielle et réduit la fréquence des données à quelques kHZ.
- Le DAQ (système d'acquisition des données) est un réseau Gigabit Ethernet qui permet le transfert et le routage des données provenant du détecteur au stockage permanent, en passant par les PCs de la ferme (c'est sur ces PCs que tournent le HLT et donc que la sélection des évènements les plus importants se fait).
- Le TFC (système de synchronisation) a pour charge de synchroniser tous les modules et de transmettre la décision du L0 Trigger aux modules du détecteur. Si

la décision du L0 trigger est positive, les données relatives à cet événement continuent leur transition vers le système d'acquisition des données.

- Le ECS (système de contrôle) a la responsabilité de piloter le détecteur. Cette tâche comprend la configuration et la supervision du détecteur, des systèmes TFC et DAQ. Le système de contrôle LHCb (voir Figure 2) repose sur PVSS, un système SCADA. Tous les modules contrôlables (environ 500,000) de l'expérience doivent être représentés dans PVSS. La modélisation des états du détecteur (et par conséquent de ses différents éléments) se fait grâce à une machine à états (FSM). La communication entre PVSS et le hardware s'appuie sur DIM. Les informations concernant la configuration sont stockées dans une base de données Oracle, la LHCb CIC DB. Cette dernière contient toutes les informations liées aux paramètres de configuration des différents modules, à la topologie des sous-systèmes (comment les modules sont-ils connectés ?) et à l'inventaire/historique du hardware. Il est donc important que le système de contrôle s'appuie sur des outils « autonomics » pour configurer et superviser l'expérience étant donné la grande quantité de modules à configurer et le nombre de tâches à exécuter.

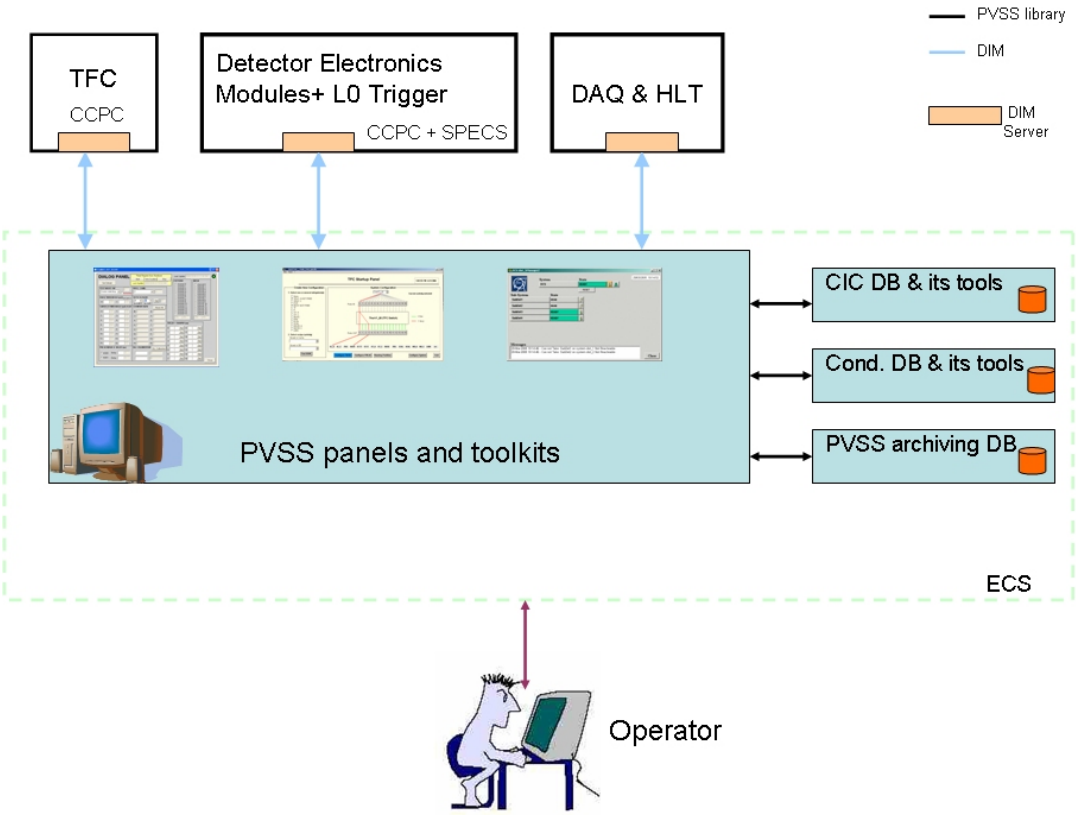


Figure 2. Architecture logicielle du système de contrôle PVSS.

Une fois le détecteur construit et le système Online fonctionnel, les expériences de physique peuvent débuter. Pour ce faire, chaque expérience physique se caractérise par un mode et une partition. Un mode correspond à définir la configuration du détecteur, selon ce que l'on veut mesurer. Par exemple, le mode PHYSICS est un mode destiné à tout mesurer (énergie, moment et traces de tous types de particules de type B mésons), alors que le mode COSMICS est utilisé pour mesurer le bruit (rayons cosmiques). La partition quant à elle, consiste à définir quelles parties du détecteur vont participer à l'acquisition des données. Ce concept est très important car il permet de faire tourner plusieurs parties du détecteur en parallèle avec des configurations

différentes. Cela facilite aussi le débogage des sous-systèmes. Faire tourner le détecteur pour acquérir des données s'appelle un *run* dans le jargon PHE. En général, un *run* dure quelques heures.

Jusqu'à présent, nous avons décrit les systèmes qui permettent d'acquérir, de sélectionner et de stocker les données les plus intéressantes.

Ces données ainsi sauvegardées sur cassettes vont faire l'objet d'analyses et de reconstructions des interactions. C'est le système Offline qui en a la charge. Ce système inclut aussi les simulations de données afin de valider les algorithmes de reconstruction et d'analyse. Ce système fait grand usage des grilles de calcul pour la reconstruction des événements et autres études physiques impliquant des opérations complexes et coûteuses.

Chapitre 2 Configurer l'expérience LHCb

L'expérience LHCb comprend différents types de modules dont la quantité varie d'un sous-système à l'autre. Il y a environ 500,000 modules à configurer et la quantité d'informations nécessaires pour les configurer peut aller de quelques kB à quelques MB.

Qui plus est, la configuration des paramètres de certains modules (coefficient de calibrations par exemple) nécessite une connaissance de la topologie du sous-détecteur telle que déterminer quelle LED illumine quelle cellule dans le détecteur HCAL. Ainsi les informations relevant des paramètres de configuration et de la topologie peuvent être combinées pour configurer un sous-système.

Outre les éléments du détecteur, le réseau du DAQ et ses centaines de routeurs doivent aussi être configurés. Cela implique notamment la création de tables de routage (IP), la configuration des fichiers des serveurs DHCP et DNS.

Le commutateur central du système TFC doit lui aussi être programmé en fonction de la partition choisie.

Décrire la topologie du système permet aussi de tester les liens et de détecter les erreurs lors de transmission de signaux entre deux modules. La description peut se faire entre les modules ou bien entre les composants de modules. Il y a donc deux niveaux de topologie : celui qui décrit les liens entre les modules, dit macroscopique et celui qui décrit la topologie du module lui-même dit microscopique.

De plus, comme la durée de vie du détecteur est longue (environ 10-15 ans), il y a fort à parier que certains éléments vont se dégrader au bout d'un certain temps, notamment pour les modules situés dans la zone de radiation. Ces derniers vont devoir être remplacés. Il est donc essentiel de savoir quels sont les modules à remplacer, les modules en réserve selon leur type et autres informations liées à la gestion du matériel.

Configurer les éléments est une étape, mais s'assurer que la configuration s'est bien déroulée en est une autre. La stratégie appliquée au sein de l'expérience LHCb est la vérification systématique des valeurs des paramètres de configuration telles qu'elles sont lues dans le hardware. Ensuite ces valeurs sont comparées à celles qui ont été écrites. La configuration s'est bien passée s'il y a concordance entre les valeurs, sinon l'expérience passe à l'état « ERROR ».

Toutes les informations qui sont donc nécessaires à la configuration de l'expérience doivent être stockées dans la LHCb CIC DB.

Il est inenvisageable d'accomplir les différentes tâches de configuration manuellement. D'une part, la quantité de travail serait énorme et fastidieuse, d'autre part, il y a de fortes chances que des erreurs d'inadvertance soient commises. Par conséquent, il est nécessaire que des outils autonomes et dynamiques (*autonomiques*) soient implémentés (là où cela est possible) et intégrés à l'architecture logicielle du système.

Chapitre 3 L'architecture logicielle

L'architecture implémentée pour permettre la configuration de l'expérience est une architecture 3 tiers (voir Figure 3). Les 3 couches sont les suivantes:

- La couche base de données permet de stocker de façon persistante les informations relatives aux paramètres de configuration, à la topologie des sous-systèmes et à l'inventaire des modules. Toutes ces informations sont dans la LHCb CIC DB, une base de données Oracle relationnelle. Le schéma de la LHCb CIC DB joue un rôle très important et se doit d'être complet afin de répondre aux besoins des utilisateurs. Des applications PL/SQL ont aussi été implémentées pour faciliter certaines mises à jour des données et pour éviter d'encapsuler de longues requêtes SQL.
- La couche traitement des données (couche objet) fait le lien entre les couches. Elle est composée d'une librairie PVSS qui permet de sauvegarder et de télécharger des paramètres de configuration. Deux scripts en Perl ont été implémentés afin de créer les fichiers de configuration des serveurs DHCP et DNS. Enfin une librairie en C, CIC_DB_lib a été écrite afin que des utilisateurs non experts en bases de données puissent manipuler de façon consistante et sûre, les informations liées à la topologie des sous-systèmes et à l'inventaire. Cette librairie inclut deux extensions, une en Python et une en PVSS.
- La couche présentation des données repose sur des interfaces graphiques PVSS et sur CDBVis, un outil en Python qui permet la visualisation de la topologie des systèmes et autres informations. Cet outil est très utile pour détecter les erreurs liées à la topologie. Tous ces panneaux utilisent ou la librairie en PVSS pour les paramètres de configuration ou la CIC_DB_lib (extensions incluses) pour communiquer avec la LHCb CIC DB. C'est cette couche qui permet l'intégration dans le système de contrôle LHCb via les panneaux PVSS.

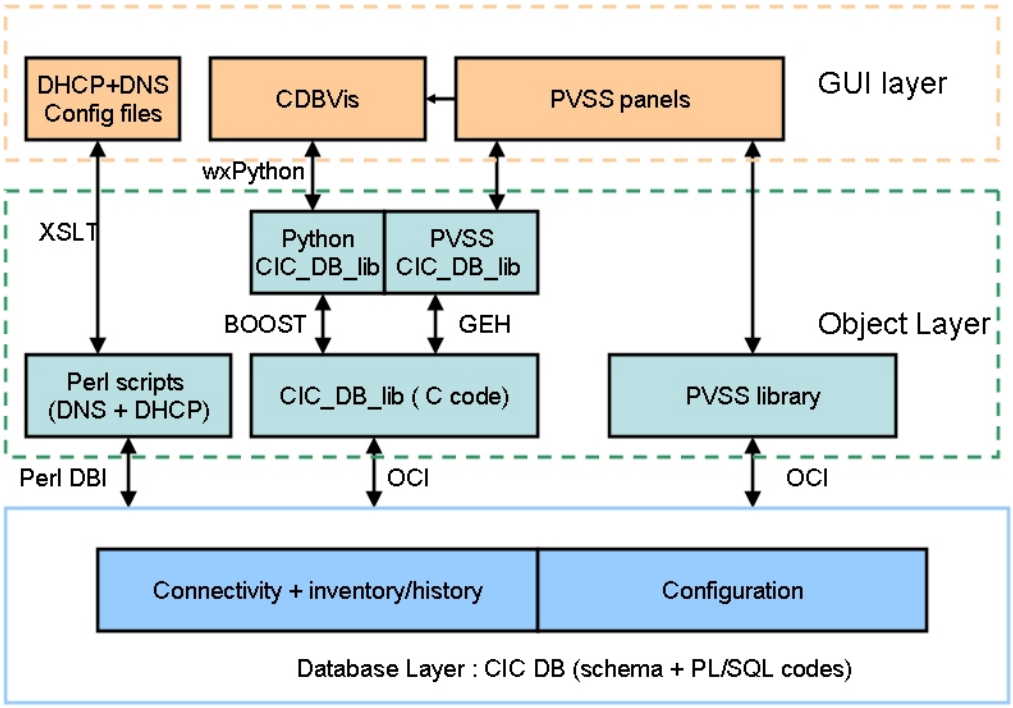


Figure 3. Architecture 3 tiers pour la configuration de l'expérience LHCb.

Cette architecture inclut autant que faire ce peu des outils « autonomics », notamment pour la création de tables de routage et destination et pour une mise à jour cohérente des informations relatives à l'inventaire. Tous les outils ont été conçus de manière à minimiser l'intervention humaine et à être robuste face à de mauvaises manipulations des données.

Chapitre 4 Besoins et cas d'usages

La méthodologie adoptée pour implémenter l'architecture 3 tiers est la suivante :

- Identification des utilisateurs de la LHCb CIC DB (sous détecteurs, DAQ, TFC et ECS) ;
- Liste des besoins (par rapport aux groupes d'utilisateurs, à l'intégration dans ECS, aux interfaces graphiques et à la sécurité) ;
- Liste des cas d'usages répartis par type d'information (paramètres de configuration, topologie des systèmes et inventaire/historique). Chaque groupe d'utilisateur nous a fait part de ses différents cas d'usages ;
- Assimiler l'environnement LHCb afin de mieux comprendre les cas d'usage et leurs implications ;
- Concevoir le schéma de la LHCb CIC DB ;
- Implémenter les outils qui permettent d'interagir avec la LHCb CIC DB et les intégrer dans le système de contrôle ;

Des besoins ont été émis par rapport au schéma des tables. Ce dernier doit être complet et consistant. Il doit aussi assurer une bonne performance. Au démarrage, le chargement d'une configuration de toute l'expérience (de la LHCb CIC DB vers le hardware) doit s'effectuer entre 5 et 10 minutes. Le schéma doit être extensible afin de pouvoir être réutilisé pour d'autres expériences futures. Enfin et non des moindres,

le schéma doit être générique, en d'autres termes il doit pouvoir s'appliquer à n'importe quel sous-système.
En ce qui concerne les besoins par rapport aux outils, un des aspects essentiels est l'obligation d'implémenter des outils qui ne requiert aucune connaissance du schéma des tables et du langage SQL. De plus, il doit être possible de manipuler et d'insérer n'importe quel type d'information contenue dans la LHCb CIC DB depuis PVSS. Une API complète a donc dû être définie en s'appuyant sur les cas d'usages. Un outil de visualisation doit être implémenté pour vérifier la topologie des systèmes qui a été insérée dans la LHCb CIC DB. Le maximum d'outils doit être « autonomics » afin de réduire la probabilité d'erreur et le nombre de tâches fastidieuses à exécuter.

Les cas d'usages ont été définis au travers de plusieurs réunions avec les différents groupes d'utilisateurs de la LHCb CIC DB. Ils ont permis de concevoir le schéma des tables et l'API pour les librairies PVSS (paramètres de configuration) et CIC_DB_lib.

Chapitre 5 Le schéma des tables de la LHCb CIC DB

Le schéma des tables a été conçu en utilisant le modèle relationnel. Chaque cas d'usage a été repris et analysé. Les entités, leurs attributs ainsi que les relations ont pu être définis.

Le choix du modèle relationnel s'explique avant tout par le fait qu'il répondait aux besoins de l'expérience et qu'un autre modèle tel que le modèle objet n'aurait fait qu'augmenter la complexité du schéma sans apporter aucun gain.

Lors de l'analyse des cas d'usage, deux niveaux de vue ont dû être distingués, le niveau hardware et le niveau fonctionnel. Autrement dit, la carte qui porte le numéro de série XDHFFD41 (hardware) accomplit les tâches de MUON_TELL1_12 (fonctionnel). Si MUON_TELL1_12 a un problème, la carte du point de vue élément physique sera remplacée. Et donc par la suite, c'est une carte qui porte le numéro de série GT54RVSS, par exemple, qui accomplira les tâches de MUON_TELL1_12. Le nom fonctionnel reste inchangé au cours du temps.

La Figure 4 montre le schéma des tables obtenu pour la modélisation des informations de l'inventaire et de l'historique. Les tables FUNCTIONAL_DEVICES et HARDWARE_DEVICES contiennent respectivement les informations fonctionnelles d'un module et les informations hardware d'un module. Sur le même principe, les tables FUNCTIONAL_COMPONENTS et HARDWARE_ COMPONENTS contiennent respectivement les informations fonctionnelles d'un composant d'un module et les informations hardware d'un composant d'un module.

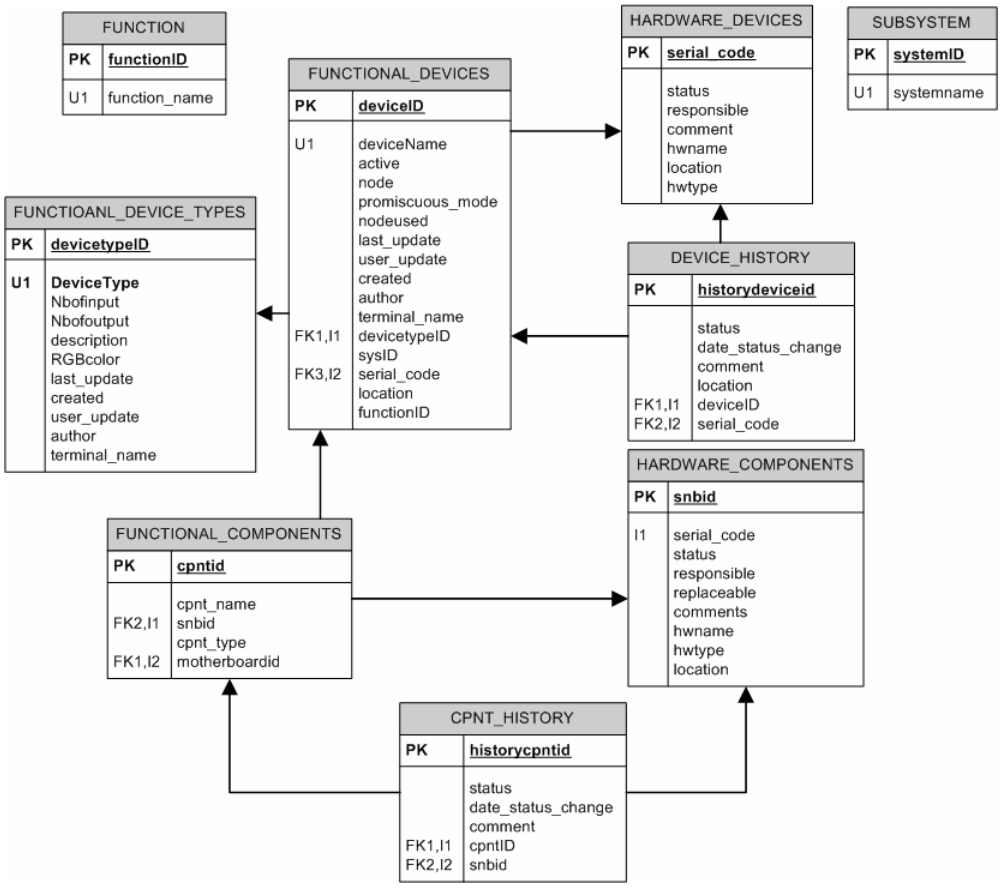


Figure 4. Schéma des tables obtenu pour la modélisation de l'information relative à l'inventaire.

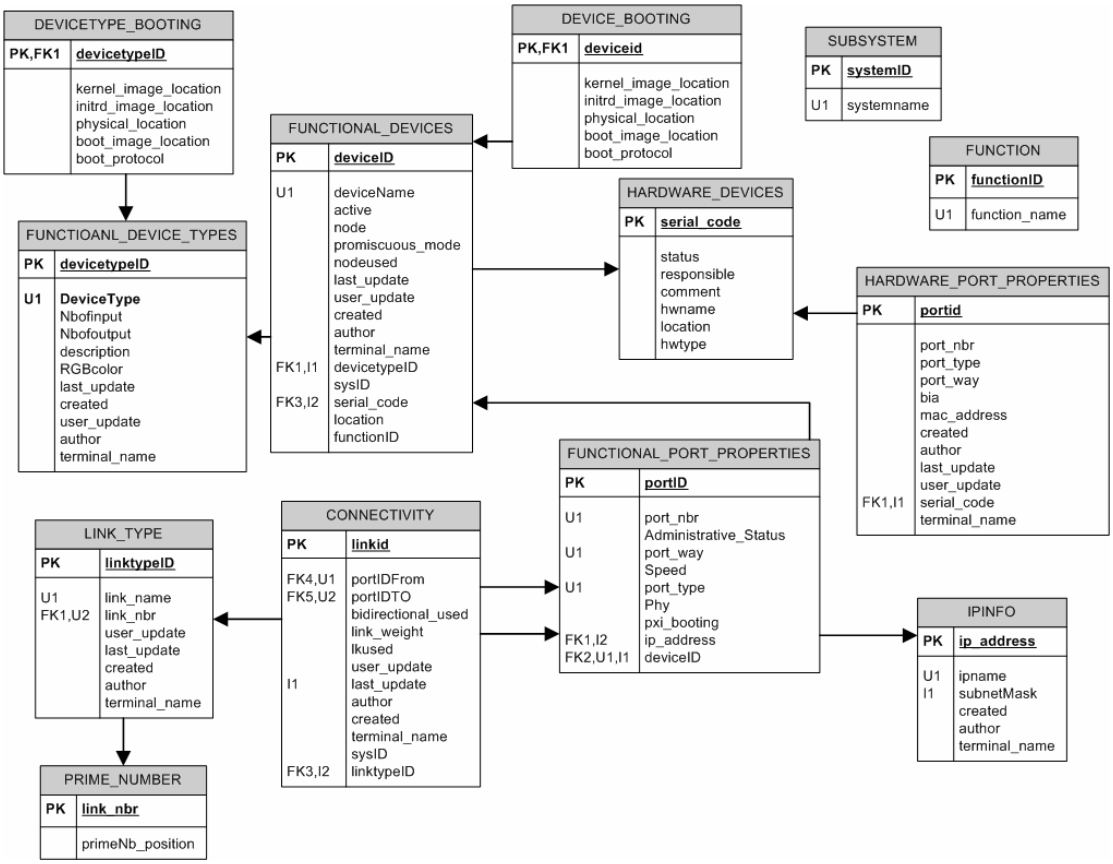


Figure 5. Schéma des tables obtenu pour la représentation de la topologie des systèmes.

Un autre aspect qui a été traité dans ce chapitre est une méthode qui permet de modéliser une relation M:N où une des deux entités ne peut avoir au plus que $2.30*10^{34}$ valeurs possibles, en utilisant des nombres premiers. Ainsi cette méthode évite la création d'une autre table. Le principe de cette méthode est d'attribuer un nombre premier à l'entité qui a le moins de valeurs possibles (appelons cette entité X et l'autre Y), en guise d'identifiant (clé primaire). Ensuite nous ajoutons un attribut à Y qui va modéliser la relation M:N en lui affectant le produit des identifiants de X (avec qui Y a une relation). Considérons les entités « employé » et « projet ». Supposons qu'il y ait au plus 50 projets en cours. L'entité « projet » est définie par un ID et un nom. L'entité « employé » est définie par un ID et un nom. Comme l'entité « projet » a le moins de tuples, nous affectons à chacun des ID un nombre premier (2 pour le projet Data Management, 3 pour le projet STAR_PHONE, 5 pour le projet JAVA_3D, etc.) Ensuite, nous ajoutons un attribut part_project à l'entité « employé ». Si par exemple, un employé travaille à la fois sur le projet Data Management et sur le projet STAR_PHONE, l'attribut part_project de cet employé vaudra 5*2=10. Et inversement, grâce à la décomposition unique d'un nombre en nombres premiers, nous retrouvons que si l'attribut part_project d'un employé vaut 10, cela veut dire qu'il travaille sur STAR_PHONE car mod(10, 5)=0 et sur Data Management car mod(10,2)=0. (mod correspond à l'opérateur modulo).

Cette méthode a été appliquée pour la relation N :M entre FUNCTIONAL_DEVICES et SUBSYSTEMS, pour celle entre FUNCTIONAL_DEVICES et FUNCTIONS et enfin pour celle entre CONNECTIVITY et LINK_TYPE, en se référant à la Figure 4 et à la Figure 5.

En reprenant les cas d'usages, nous avons pu vérifier que le schéma des tables était complet.

Chapitre 6 Création automatique de tables de routage et de destination avec PL/SQL

Outre le schéma des tables, la couche accès aux données comprend des applications PL/SQL. L'une d'entre elles (**routingtable_pck**) permet la création et la mise à jour automatique des tables de routage et de destination.

Ces deux types de table permettent de déterminer le chemin à prendre en fonction de la destination. Il a fallu définir la notion de destination. Certains modules ne peuvent pas être des destinations. Un commutateur par exemple ne peut jamais être une destination puisqu'il ne peut pas recevoir de paquets qui lui soient adressés. Par contre, un PC de la ferme ou une carte de type TELL1 sont des destinations parce qu'ils peuvent envoyer ou recevoir des paquets.

Ainsi nous modélisons la topologie d'un système à l'aide de deux types de nœuds, des nœuds intermédiaires et des nœuds terminaux. Par définition, un chemin est une succession de noeuds. Un chemin de routage est une succession de nœuds où le seul nœud terminal prend la dernière position dans la séquence. Il est à noter qu'une table de routage est toujours créée pour un commutateur ou un routeur, i.e. des nœuds intermédiaires. Et, donc un chemin de routage commence toujours par un nœud intermédiaire (un commutateur ou un routeur).

Afin de faciliter la recherche de chemins de routage, un système de poids est appliqué aux nœuds (0 si c'est un nœud intermédiaire, 1 si c'est un nœud terminal) ainsi qu'aux liens (0 si c'est entre deux nœuds intermédiaires, 1 si c'est entre un nœud terminal et

un nœud intermédiaire, 2 si c'est entre un nœud intermédiaire et un nœud terminal, 3 si c'est entre deux nœuds terminaux).

En outre, étant donné que trouver tous les chemins est un problème NP complet, nous introduisons un paramètre M, caractéristique de la longueur maximale d'un chemin en fonction de la topologie du système. Dans le contexte LHCb, les topologies sont telles que ce paramètre a permis de réduire la complexité du problème. Ce paramètre est défini par l'utilisateur. Par défaut, il est égal à 10 (dans le contexte LHCb, les chemins trouvés avaient une longueur inférieure ou égale à 10).

Ainsi un chemin de longueur J est un chemin de routage de longueur J si et seulement la somme des poids des J-1 liens vaut 0 et la somme des poids des J liens vaut 2. Autrement dit, tous les poids des liens du chemin valent 0 sauf le dernier lien du chemin qui vaut 2.

En utilisant cette formule, nous avons développé l'algorithme dont les étapes sont détaillées ci-après. Les paramètres d'entrée sont le nom du routeur et la longueur maximale d'un chemin (M), qui est optionnel.

1. Création d'une vue logique de la topologie du réseau, où les nœuds du graphe ne sont plus un couple (module, port) mais seulement le nom du module. Ainsi, s'il y a trois liens entre les modules A et B, dans cette vue simplifiée, nous considérons que le fait qu'il existe un lien entre le module A et le module B. En outre, un lien bidirectionnel est représenté en tant que deux liens unidirectionnels. Par exemple s'il existe un lien bidirectionnel entre les modules A et B, nous considérons ce cas de figure, dans cette représentation, comme un lien partant du module A vers B et un autre lien du module B vers le module B.
2. Ensuite, nous regroupons par paire les liens obtenus dans cette vue simplifiée et vérifiant 4 conditions. A titre de remarque, une paire de liens comporte 3 nœuds et a une longueur de deux. Une paire de liens est donc validée, si
    a. le deuxième nœud du premier lien correspond au premier lien du deuxième lien, condition qui va de soi pour assurer l'enchaînement ;
    b. le premier nœud est différent du troisième nœud afin d'éviter les cycles ;
    c. le poids du premier lien ne vaut pas deux ;
    d. les deux liens transportent des types de données compatibles. Un lien qui transporte des données relatives à des événements n'est pas compatible avec des données relatives au contrôle.
3. Puis, tous les liens qui ont un poids égal à deux et qui commencent par le routeur donné en paramètre d'entrée sont sélectionnés. Ainsi nous trouvons tous les chemins qui ont une longueur de 1.
4. Une boucle « while » (voir Figure 6) portant sur la longueur du chemin *i* est exécutée. La condition d'arrêt est soit la longueur du chemin dépasse M, soit tous les chemins ont été trouvés. Elle inclut l'exécution des instructions suivantes :
    a. Ajout d'un nœud compatible aux chemins précédemment trouvés (vérification qu'il n'y a pas de cycles et que le type de données est compatible) ;
    b. Le poids du chemin et le type de données que peut transporter le chemin sont remis à jour. S'il existe un chemin dont le poids vaut 0, nous continuons, sinon nous stoppons la boucle.

A chaque itération i, nous trouvons les chemins de routage de longueur i.

5. A ce niveau, nous avons trouvé tous les chemins de routage. Il nous faut revenir au concept de port. Nous sélectionnons le chemin le plus court (un flag est mis à 1 signifiant que c'est ce chemin qui a été choisi) et faisons la correspondance avec les identifiants de ports afin de retrouver par la suite les adresses IP et MAC.
6. Nous insérons la table de routage ainsi obtenue dans la table (table au sens base de données).

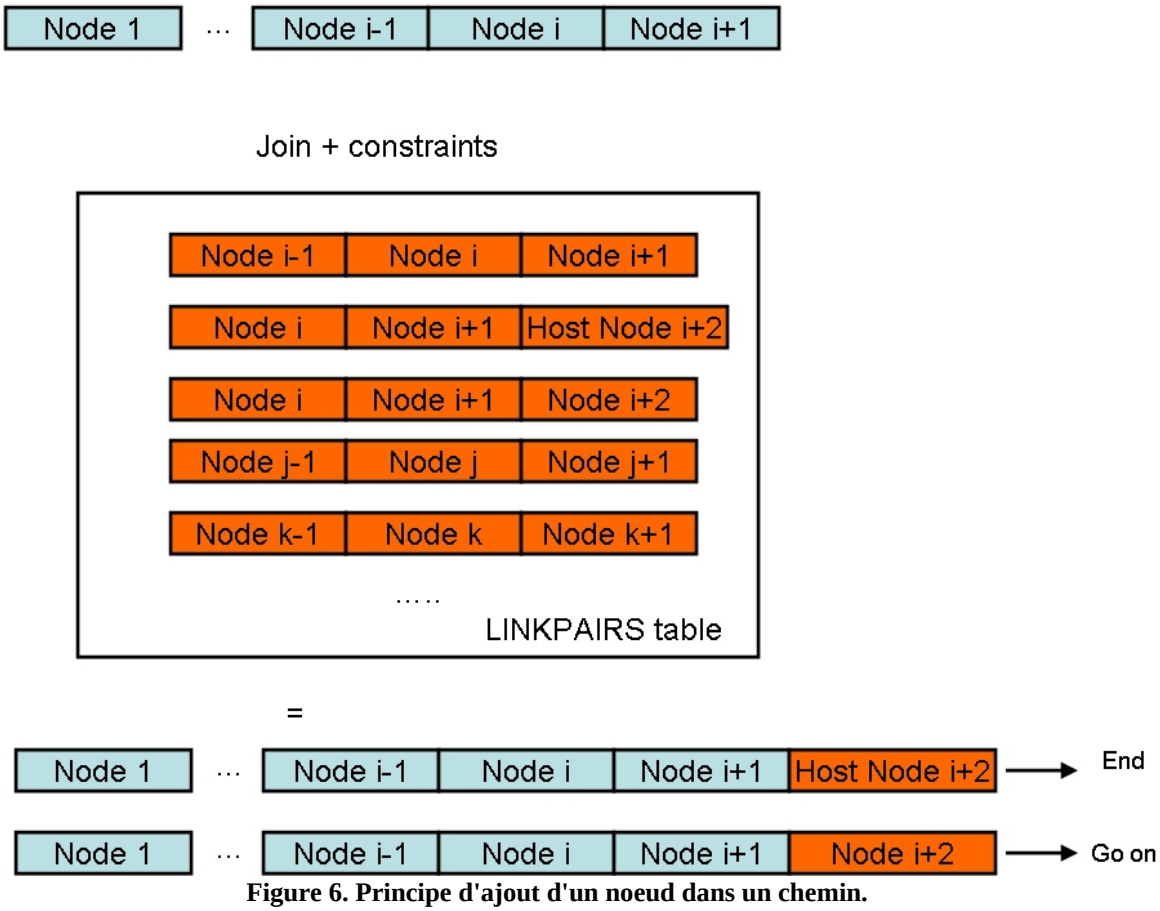


Figure 6. Principe d'ajout d'un noeud dans un chemin.

Par une démonstration par l'absurde, nous prouvons que notre algorithme trouve bien tous les chemins de longueur<=M.

Les tables de destination utilisent le même principe. La différence réside dans le fait que les tables de destination peuvent être créées pour un nœud terminal (ce qui n'est pas le cas pour des tables de routage) et que nous ne sélectionnons pas les plus courts chemins (étape inutile).

Cet algorithme a été déployé sous forme d'un paquetage PL/SQL, langage qui est spécifique aux bases de données Oracle. L'avantage de ce langage est qu'il est exécuté du côté serveur et qu'il peut être encapsulé dans n'importe quel autre langage (C, Perl, Python, java, etc.). Cela évite le transfert de longues lignes de requêtes SQL à travers le réseau.

Cependant une version en C de cet algorithme a aussi été écrite. Elle est utilisée pour répondre aux questions du type, donner tous les chemins entre un module A et un module B, ou de façon plus générale, donner tous les chemins appartenant à tel système, et passant par le module A. Nous avons préféré exécuter ce code du côté client étant donné que les requêtes sont dynamiques, ce qui n'est pas le cas des tables

de routage devant être créées pour tous les routeurs du système DAQ, et des tables de destination devant être créées pour le commutateur TFC et pour tous les serveurs DHCP du system DAQ.

Les tables de destination sont utilisées pour le commutateur TFC et pour les serveurs DHCP.

Avec ce paquetage PL/SQL, des applications « autonomics » peuvent être implémentées afin d'éviter à l'utilisateur une programmation manuelle des centaines de tables de routage et des mises à jour en cas de modifications dans le réseau, telle qu'un ajout de PC ou une panne d'un routeur.

Chapitre 7 La couche traitement des données

Cette couche comprend divers types de librairies qui ont été écrits pour interagir avec la LHCb CIC DB. Ils ont été implémentés dans l'optique de faciliter le déploiement d'outils « autonomics » qui utilisent ces librairies:

- Deux scripts Perl permettent la création automatique de fichiers de configuration pour les serveurs DHCP et DNS. Pour mettre au point le fichier de configuration du serveur DHCP, sa table de destination est d'abord créée afin de déterminer les éléments qui seront configurées par ce dernier. Ensuite pour chacune des destinations, nous retrouvons son adresse IP et MAC, son nom IP et enfin son fichier pour démarrer (contenant sa configuration). Les résultats des requêtes SQL sont encapsulés dans XML. Le deuxième script Perl créée les fichiers de configuration pour le serveur DNS (correspondance entre adresses IP et noms IP). XML est aussi utilisé pour encapsuler les résultats des requêtes SQL. Nous montrons un exemple de dispositif « autonomics » utilisant PVSS qui permet de mettre à jour les fichiers de configuration des serveurs DHCP et DNS suite à une modification du réseau (voir Figure 7).

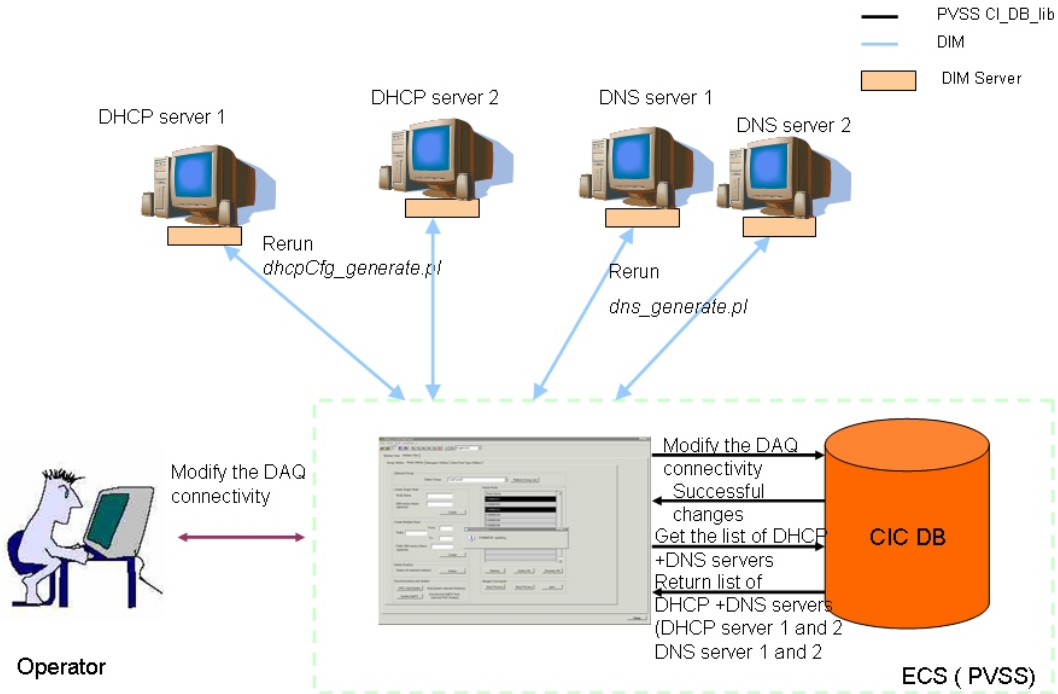


Figure 7. Exemple de dispositif « autonomics » pour mettre à jour les fichiers de configuration des serveurs DHCP et DNS, en utilisant PVSS et la LHCb CIC DB.

- Une librairie en C, CIC_DB_lib qui permet d'insérer, de mettre à jour, d'effacer et de retrouver des informations relatives à la topologie des sous-systèmes et à l'inventaire. OCI est utilisé en tant qu'interface DB. Pour assurer la consistance des données en matière d'inventaire, des diagrammes de cas d'usage (un exemple est montré par la Figure 8) ont été conçus. Ils ont permis notamment de définir les mises à jour nécessaires à la suite de chaque changement de statut d'un module. Deux extensions ont été écrites une en Python (via BOOST) et une en PVSS (via GEH). Une API a été créée afin de déterminer la liste des fonctions requises par les utilisateurs pour interagir avec la LHCb CIC DB. Cette API a été développée en utilisant les cas d'usage définis au chapitre 4. Ainsi CIC_DB_lib fournit toutes les fonctions nécessaires à l'utilisateur pour retrouver, insérer ou modifier les données de manière sûre et cohérente. CIC_DB_lib a été écrite dans l'optique de faciliter la mise en place d'outils « autonomics » qui l'utilisent, en réduisant les interventions humaines. Si nous pouvons déterminer à l'avance les conséquences d'une telle modification, nous l'implémentons directement sans demander à l'utilisateur de le faire. C'est le cas par exemple de mettre un commutateur h en mode inactif. Tous les liens de ce dernier seront automatiquement mis en mode inactif. De même pour les chemins de routages.

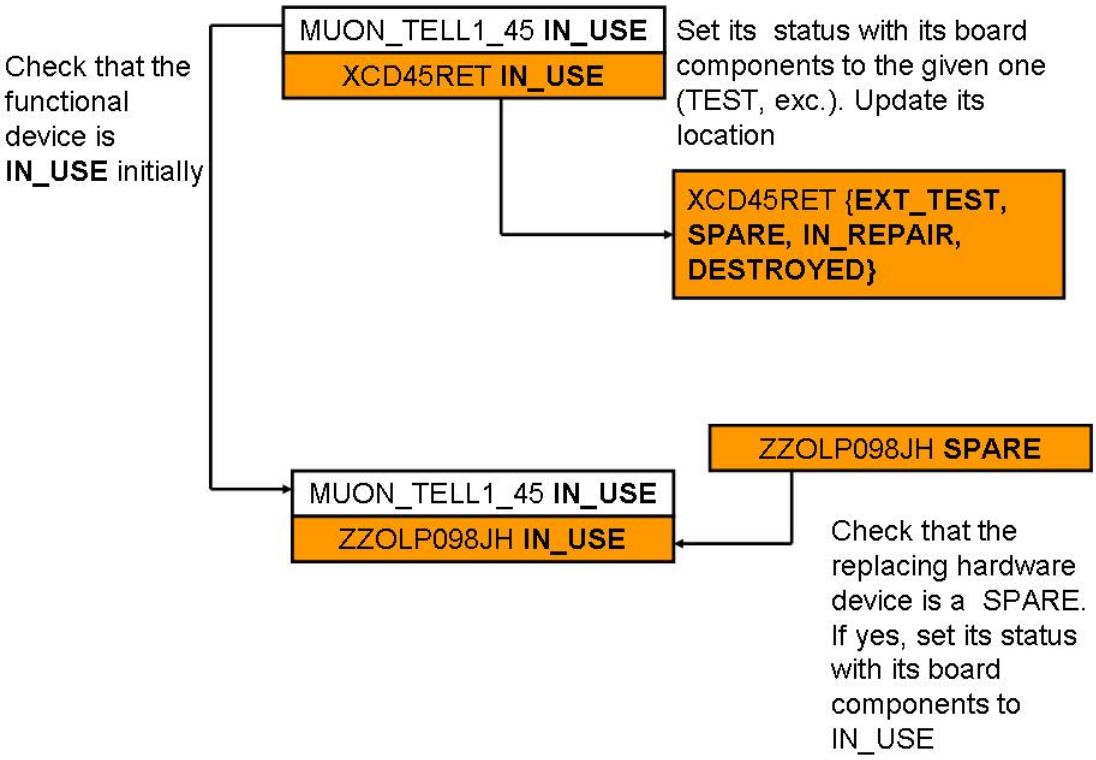


Figure 8. Exemple d'un diagramme dessiné pour illustrer le cas d'usage "remplacer un module".

- Une librairie en PVSS permet de sauvegarder et de télécharger des configurations, aussi appelées recettes. L'utilisateur définit les paramètres de configuration avec leurs valeurs et ce pour une liste de modules. L'ensemble (liste de modules, paramètres et valeurs de configuration par type de module) constitue une recette. Il est possible d'avoir plusieurs versions d'une même recette.

Chapitre 8 Structure de la couche présentation des données

Plusieurs interfaces graphiques ont été implémentées afin de rendre les données visibles.

- CDBVis est un outil Python qui permet d'insérer la topologie d'un sous-système et de la visualiser afin de vérifier qu'il n'y a pas d'erreur. Un exemple de capture d'écran est donné par la Figure 9. PVSS n'a pas pu être utilisé car c'est un outil inadéquat pour afficher des liens entre modules. Toutes les requêtes vers la LHCb CIC DB s'appuient sur l'extension Python de CIC_DB_lib.

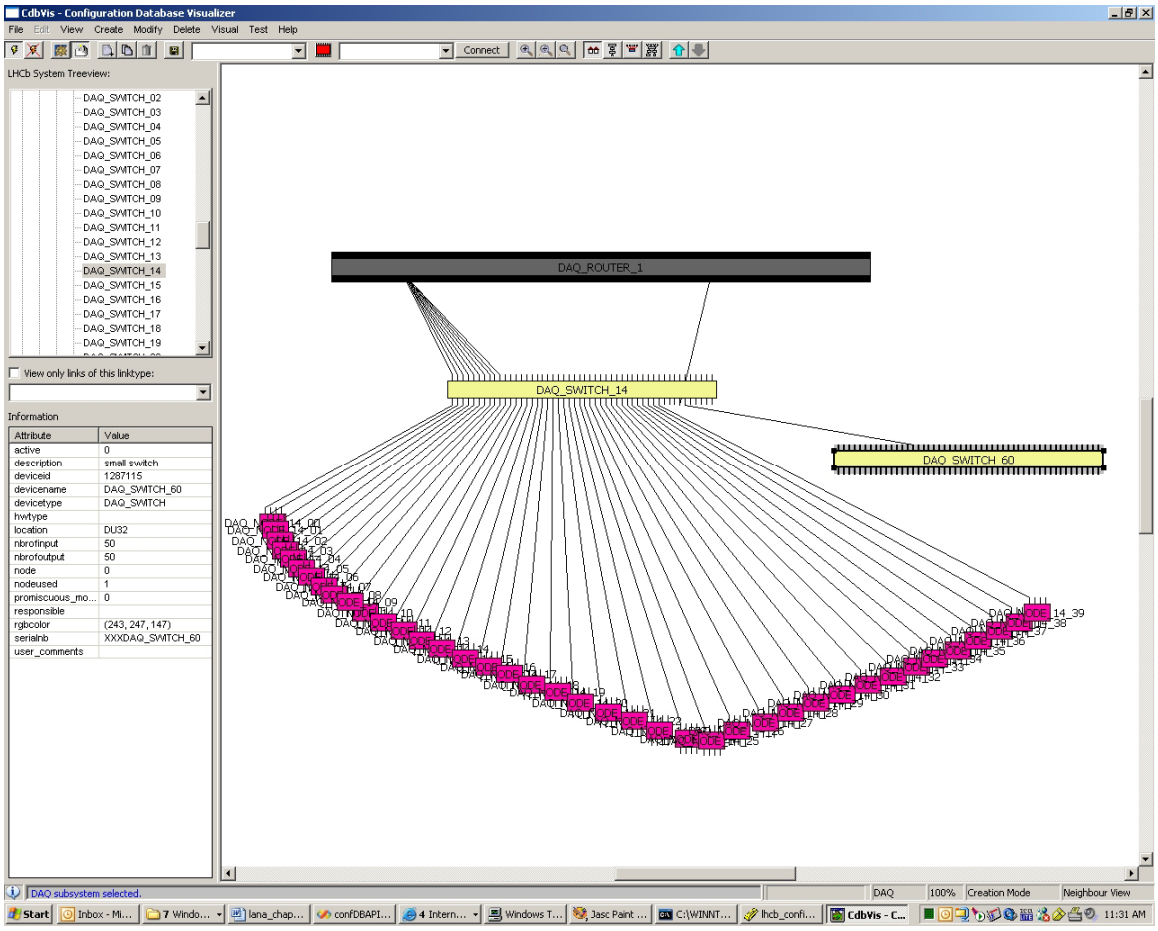


Figure 9. Capture d'écran de CDBVis affichant les liens entrants et sortants du DAQ_SWITCH_14 (un commutateur appartenant au réseau DAQ).

- Les fichiers de configuration pour les serveurs DNS et DHCP ont été créés à partir des fichiers XML en utilisant XSLT. Cela permet si besoin de convertir les fichiers XML en d'autres formats de fichiers tels que HTML.
- Des interfaces graphiques PVSS définies par l'utilisateur permettent de sauvegarder et de télécharger des recettes pour un module ou un groupe de modules, de superviser les modules. Elles permettent aussi télécharger et de visualiser des tables de routage pour un commutateur, d'obtenir des informations sur la topologie des systèmes pour la configuration de certains paramètres et de programmer le commutateur TFC en fonction de la partition choisie (voir Figure 10). Toutes les interfaces graphiques PVSS utilisent ou la librairie PVSS pour les recettes ou l'extension PVSS de CIC_DB_lib pour communiquer avec la LHCb CIC DB. Il est aussi important de noter que dans les interfaces graphiques PVSS,

les informations sur les paramètres de configuration, sur la topologie des systèmes et sur l'inventaire/historique sont combinées afin de permettre un meilleur automatisme des actions.

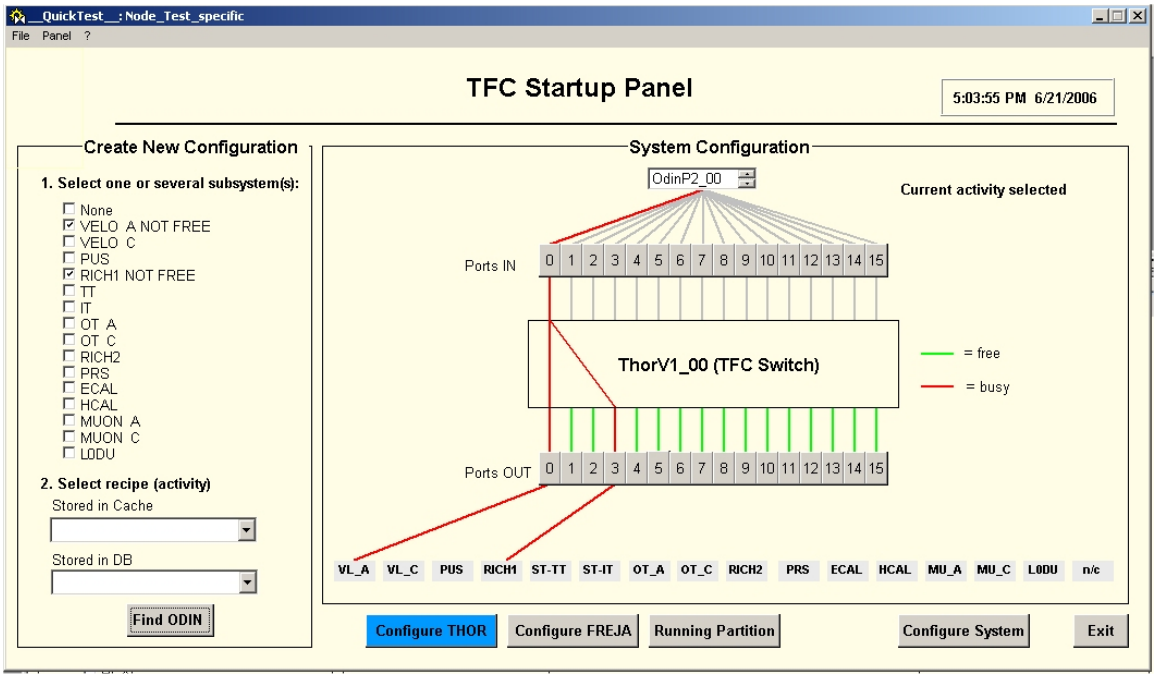


Figure 10. Exemple d'une interface graphique PVSS implémentée pour déterminer la programmation du commutateur TFC. Les systèmes VELO_A et RICH1 ont été sélectionnés. En utilisant les informations contenues dans la LHCb CIC DB, nous retrouvons les ports de sortie du commutateur TFC qui devront envoyer des signaux aux deux systèmes sélectionnés.

Chapitre 9 Validation des algorithmes de routage et de destination

Routingtable_pck, le paquetage PL/SQL qui permet entre autres de créer les tables de routage, a été appliqué sur différentes topologies de systèmes (incluant des cycles) afin d'en tester la robustesse.

Le temps d'exécution de l'algorithme dépend de la complexité de la topologie, notamment au niveau nombre de liens et de la longueur maximale du chemin.

Ainsi la table de destination du commutateur TFC est créée en 1mn 15 sec, en moyenne et 36000 chemins sont trouvés. La table de routage d'un routeur type Flow (voir Figure 5) est créée en 11 sec pour 5542 chemins et celle du DAQ_ROUTER_1, en 13 sec 45 pour 20016 chemins.

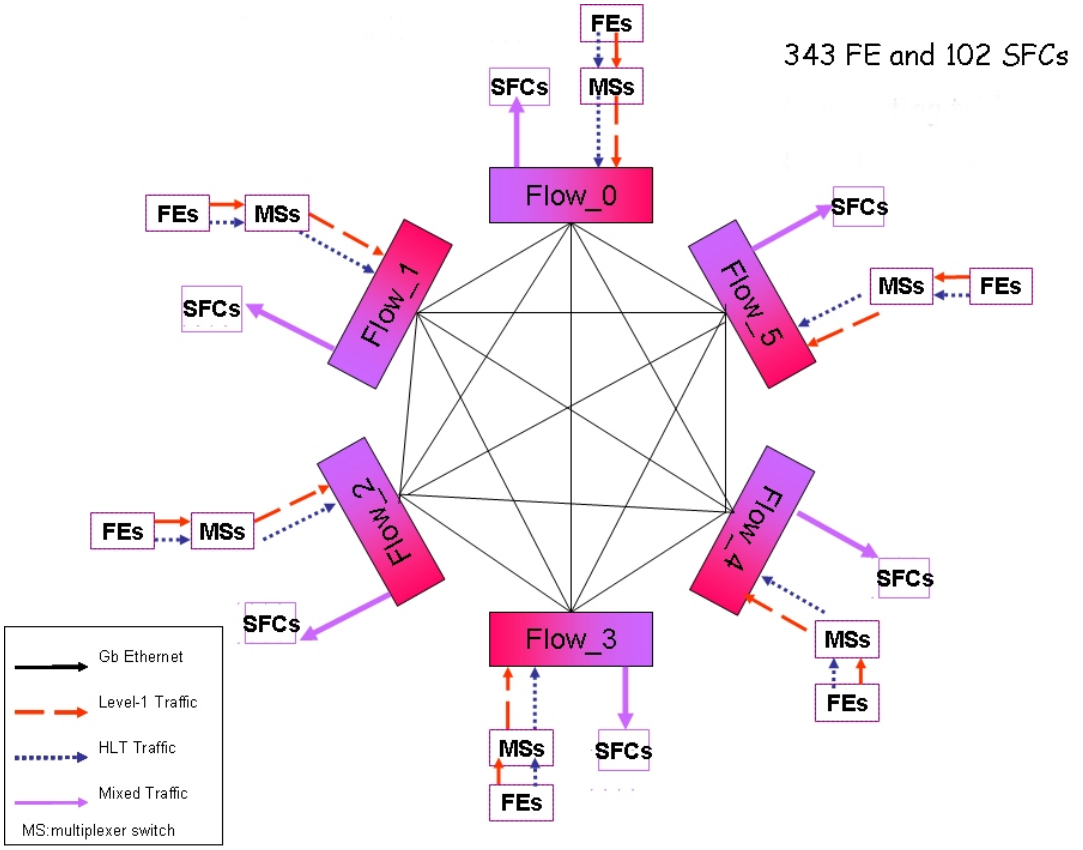


Figure 11. Topologie en forme de fleur qui avait été proposée pour implémenter le réseau DAQ.

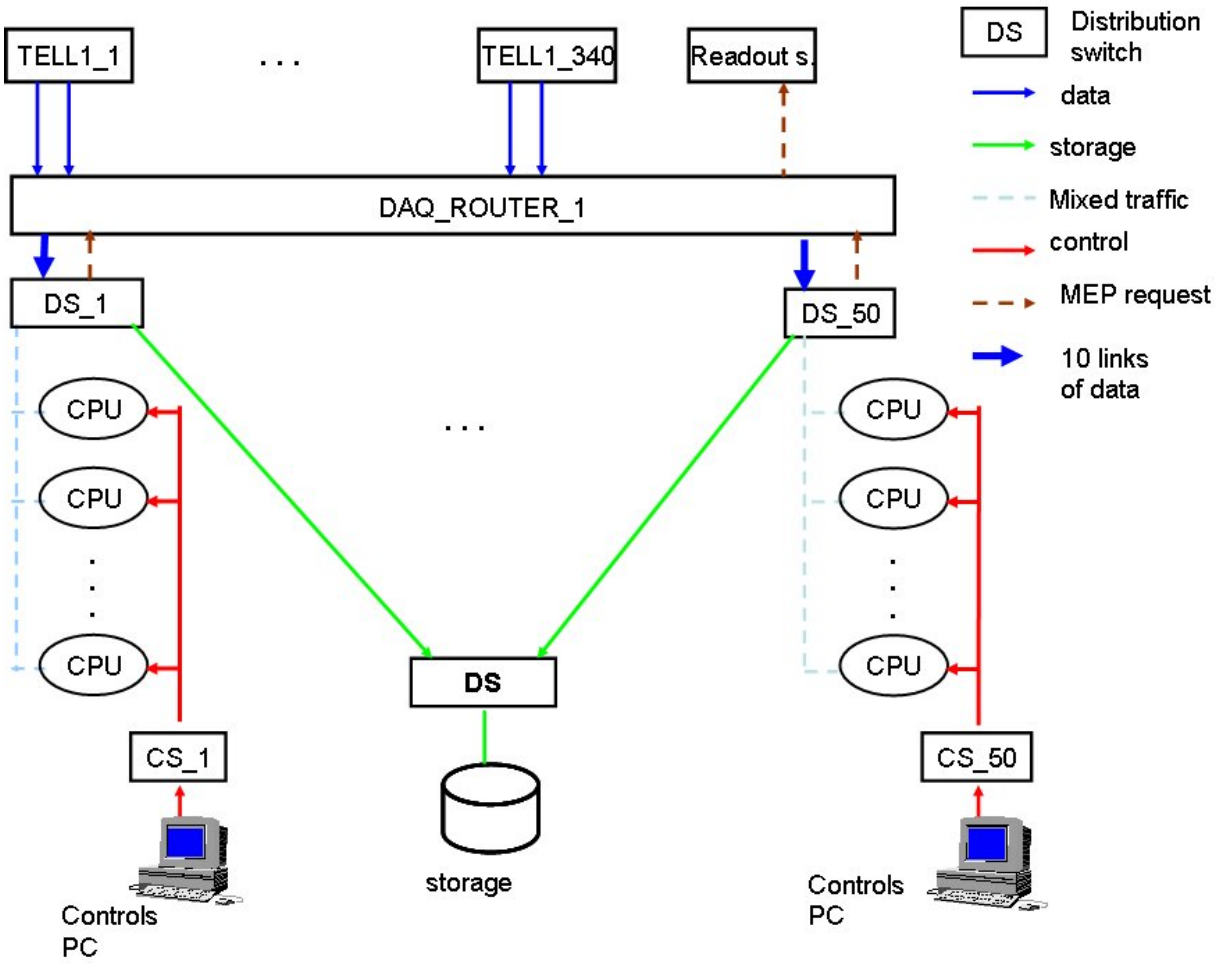


Figure 12. Topologie du réseau DAQ.

Ensuite, nous avons aussi programmé la table de routage dans le routeur. Tout d'abord la table a été téléchargée de la LHCb CIC DB vers PVSS, puis de PVSS vers le routeur en utilisant DIM (version légère de CORBA) et des sockets.

L'application permettant de créer le fichier de configuration du serveur DHCP a été testée en copiant le fichier « dhcpd.conf » sur un véritable serveur DHCP. Le fichier a été accepté et compris par le serveur DHCP.

Chapitre 10 Validation de CIC_DB_lib

Différents tests ont été appliqués à la librairie, en mettant l'accent sur les points suivants :

- les fonctions ainsi que leurs extensions faisaient bien ce qu'elles devaient faire ;
- le comportement des fonctions d'insertion, de mises à jour et de délétion en cas de mauvaise utilisation (erreur ou violation de certaines contraintes) ;
- le comportement de certaines fonctions (insertion et mise a jour) lorsqu'elles sont exécutées de façon concurrentielle ;
- Les mises à jour automatiques des tables de routage et de destination stockées dans la LHCb CIC DB.

Cette librairie a été utilisée pour insérer la topologie du sous-détecteur HCAL, comprenant environ 14 000 liens et 6231 modules.

Les chercheurs du groupe HCAL ont essentiellement fait des requêtes du type, donner tous les chemins entre une carte DAC et les modules de type « Channel ». Ils ont utilisé la version C pour l'insertion et l'extension PVSS pour faire les requêtes de chemins.

Le groupe VELO a, lui aussi, inséré une partie de sa topologie, contenant quelques topologies internes de certaines cartes (module « hybrid » par exemple). La Figure 13 montre une petite partie (10%) de la topologie du sous-détecteur VELO. Outre la topologie entre les modules, celle des cartes « hybrid » et des cartes répétiteurs doivent être aussi décrites dans la LHCb CIC DB.

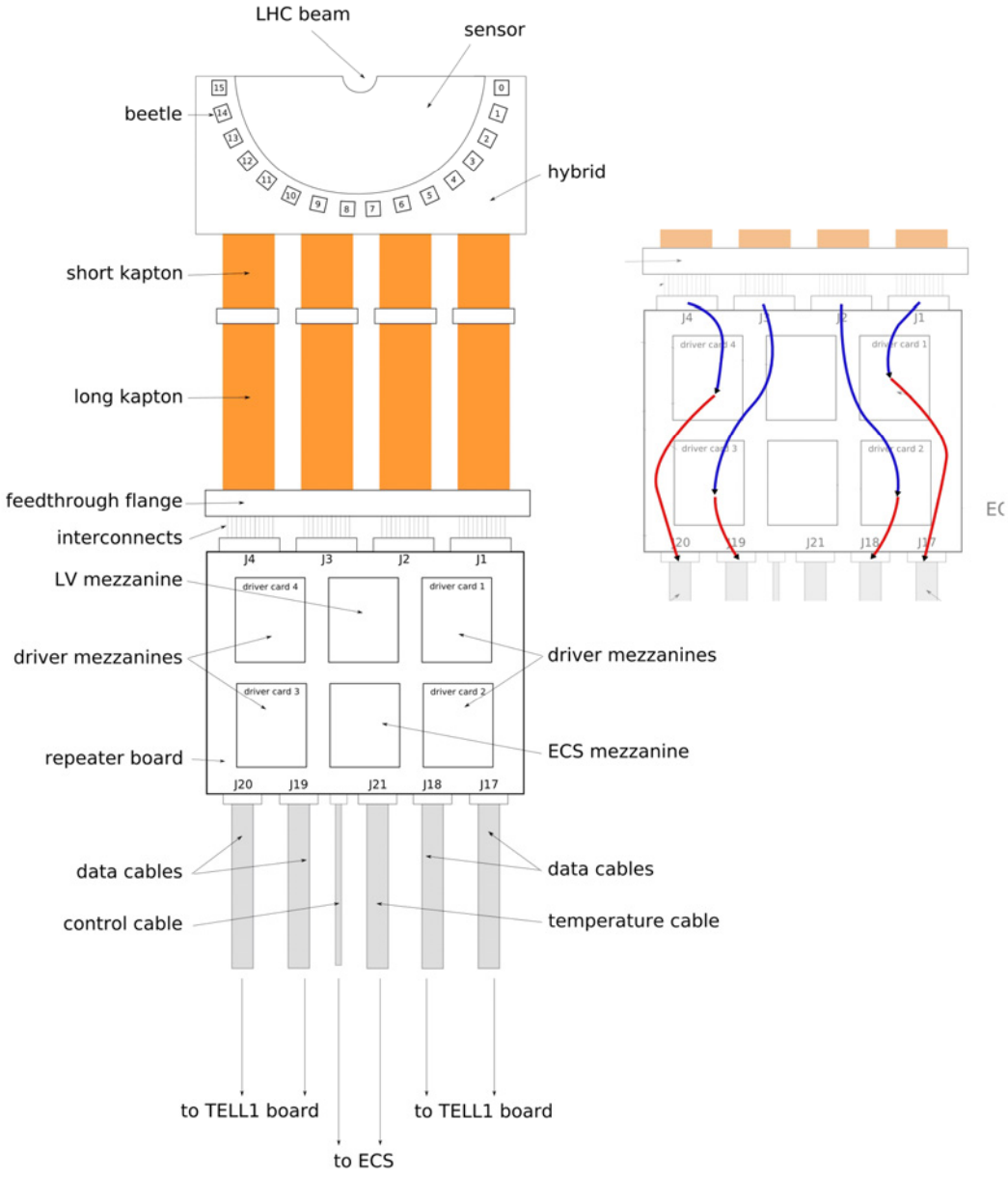


Figure 13. Extrait de la topologie du sous-détecteur VELO. La topologie des cartes "hybrid" et des cartes répétiteurs doit être décrite dans la LHCb CIC DB.

Grâce à ce système, nous avons pu tester les fonctions relatives aux informations microscopiques (liens et modules internes). L'outil CDBVis et l'extension Python de CIC_DB_lib ont été utilisés pour insérer une partie de la connectivité des chambres du système MUON.

Tous les tests ont été menés avec succès.

Chapitre 11 Autres domaines où la LHCb CIC DB peut être utilisée

Ce chapitre a pour but de présenter d'autres domaines où la base de données LHCb CIC DB peut être utilisée. Il s'agit des grands télescopes et de l'expérience des chambres de muons chez ATLAS.

Les grands télescopes doivent aussi être configurés. Nous allons brièvement présenter deux projets VLT (Very Large Project) et ALMA (Atacama Large Millimeter Array). Dans les deux projets, le groupe ESO a déployé son propre DBMS afin de configurer leur matériel.

Le VLT est constitué de 4 télescopes identiques ayant chacun un diamètre de 8 m. Ils peuvent être utilisés indépendamment ou en tant qu'un seul instrument. Les logiciels de contrôle reposent sur des terminaux et des microprocesseurs utilisant les systèmes VME. Des unités de contrôle locales supervisent une partie de l'équipement. Le VLT comprend une base de donnée qui fournit des données en temps réel et a été implantée par le groupe ; à l'époque, les temps d'écriture dans une base de données étaient bien inférieurs à leurs besoins. Cette base de donnée peut être assimilée à un ensemble de fichiers.

Chaque unité de contrôle télécharge les informations relatives à la configuration du matériel supervisé. Il n'y a pas d'information liée à la topologie des systèmes car elle est triviale dans leur cas. La base de donnée est orientée objet avec beaucoup de classes telles que moteurs, encodeurs, etc. La description d'un télescope nécessite 6400 instances, soit 6400 entrées dans la base de données.

Le projet ALMA, lui, est composé de 64 télescopes de 12 m de diamètre. Ils ont déployé un environnement d'application, ACS qui utilise CORBA. ALMA contient une impressionnante liste d'équipements et de logiciels. La configuration des télescopes change en fonction du temps. Si une antenne ne fonctionne plus, le système doit être reconfiguré. La description détaillée des télescopes est sauvegardée dans une base de données. Cette dernière est utilisée, au démarrage, pour déterminer les modules qui sont en ligne. En effet, les modules (antennes par ex.), dès qu'ils sont allumés, doivent le signaler à la base de données. La base de données compare la liste des modules en ligne avec celle qu'elle a. S'il y a un problème de correspondance, le nouveau module en ligne est ajouté. Ils utilisent donc des outils « autonomics ».

En outre, contrairement au projet précédent, l'historique des modules est stocké. Cette information est stockée dans une autre base de données relationnelle s'appuyant sur Oracle.

Nous pensons que la LHCb CIC DB et ses outils auraient pu être implémentés dans le projet ALMA. En faisant quelques adaptions, l'ensemble que nous avons développé aurait pu s'appliquer pour la partie configuration et historique. Le cas inverse n'est pas vrai, puisque les projets ALMA et VLT ne sauvegardent aucune information sur la topologie des systèmes. En outre, la dualité entre équipement hardware et équipement fonctionnel n'existe pas. Seul le niveau équipement physique est utilisé.

Un autre cas, où la LHCb CIC DB et ses outils auraient pu être utiles, est l'expérience des chambres du système MUON chez ATLAS.
Là aussi, des modules doivent être configurés et la structure des systèmes doit être décrite sous forme d'arbres. La Figure 14 montre le schéma des tables que ce groupe a obtenu. Ils ont aussi appliqué le modèle relationnel. Le concept de port n'est pas utilisé pour décrire la topologie des systèmes. Les paramètres de configurations sont stockés de manière plus ou moins semblable à celle implémentée par le groupe de support *CERN PVSS.*

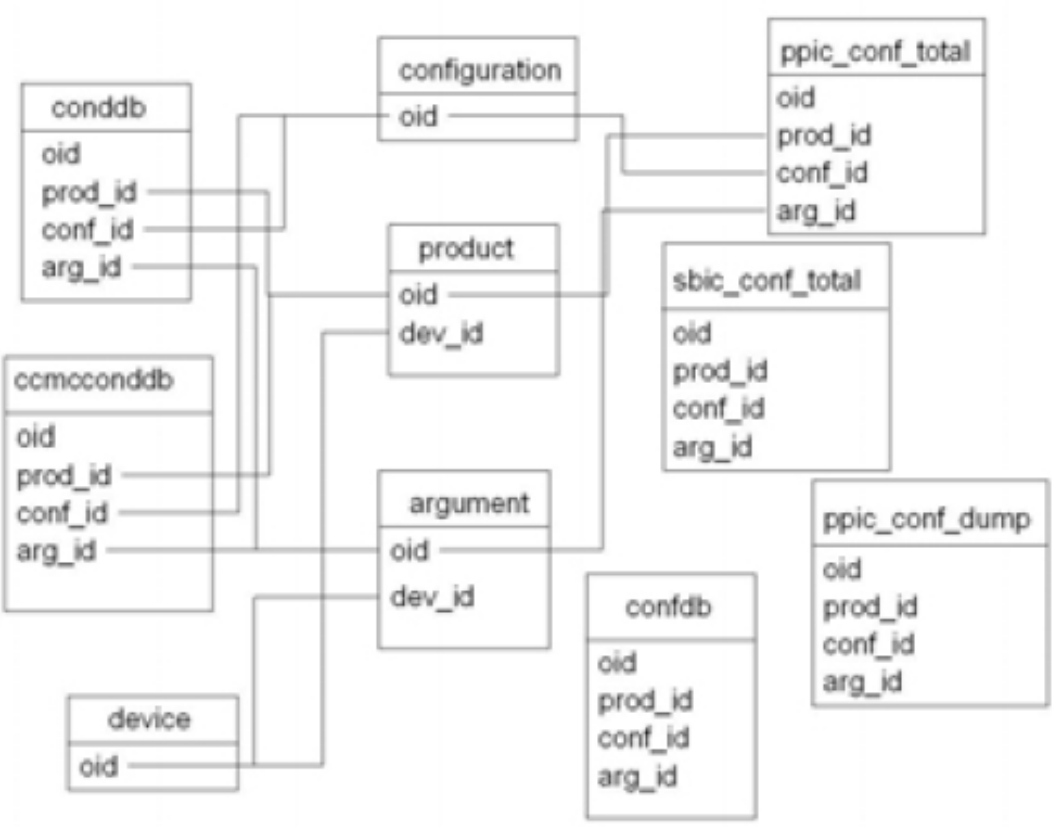


Figure 14. Schéma des tables obtenu par le groupe des chambres du détecteur MUON chez ATLAS.

Nous pensons que le schéma des tables de la LHCb CIC DB et les outils développés, moyennant sans doute quelques ajustements, auraient pu s'appliquer dans ce contexte. En tous cas, le niveau de granularité en matière de description des liens est plus fin dans la LHCb CIC DB.

Conclusion

Cette thèse propose une architecture 3 tiers qui permet de configurer l'expérience LHCb de manière « autonomics ». Cette architecture repose entre autres sur une base de données Oracle relationnelle qui contient toutes les informations nécessaires pour configurer l'expérience. Ces informations relèvent des paramètres de configuration selon le type de module, de la topologie des systèmes et de l'inventaire/historique des modules.
Une méthode utilisant les nombres premiers permet de modéliser une relation N :M où une des deux entités a au plus $2.30*10^{34}$ valeurs distinctes. Cette méthode gagne en

simplicité en évitant de créer des tables supplémentaires. Selon les cas, la performance peut être meilleure.

Routingtable_pck est un paquetage PL/SQL qui permet d'automatiser la création de tables de routage et de destination. Leurs mises à jour se font aussi automatiquement suite à un changement de topologie.

Des librairies intelligentes ont été écrites de façon à ce que des outils « autonomics » puissent être implémentés, en réduisant les interventions humaines.

Enfin des interfaces graphiques, telles que CDBVis ont été développées afin de visualiser et de modifier les informations contenues dans la LHCb CIC DB. CDBVis permet aussi par exemple de détecter les erreurs lors de l'insertion des liens dans un système.

Ce projet va suivre son cours. De nouvelles fonctionnalités vont être ajoutées aux outils. Toutes les interfaces graphiques n'ont pas pu être complètement achevées.

En outre, une extension du schéma des tables de la LHCb CIC DB peut être faite en ajoutant des informations sur les actions à prendre si telle erreur survient. En effet, quand le détecteur va se mettre en marche, des erreurs plus ou moins connues vont revenir à certaines fréquences. Ces erreurs qui se produisent un plus fréquemment que les autres (telles qu'un comportement étrange de la carte VELO_TELL1_12 après une coupure soudaine d'électricité) peuvent être stockées dans la base de donnée et on peut leur associer un ensemble d'actions que le système de contrôle (en utilisant PVSS et sa machine à états) peut prendre. Cela permet d'obtenir un système de contrôle plus robuste.

De plus, des études sur le parallélisme peuvent être menées pour améliorer la performance.

General introduction

Si l'homme parfois ne fermait pas les yeux, il finirait
par ne plus voir ce qui vaut la peine d'être regardé.
RENÉ CHAR.

The configuration of large physics experiments

The topic of this thesis is the use of autonomics in the configuration of a large physics experiment such as LHCb.

The LHCb (Large Hadron Collider beauty) experiment is located at CERN (the European Laboratory for Particle Physics) [1]. It is one of the four experiments at the LHC (Large Hadron Collider) [2].LHCb is an international collaboration containing around 500 physicists from 50 participating institutes.

The objective of LHCb is to study CP violation on B mesons [3], a necessary condition for explaining the dominance of matter over antimatter after the Big-Bang. Collisions between two circulating beams of protons will generate particles that will produce signals in the sensors of the LHCb detector.

The LHCb detector is a complicated collection of a large number (500,000) of devices. These devices fulfil many different tasks and span a vast range of technologies. They range from particle detection and identification to signal processing and data handling.

In this thesis we are mainly interested in the configuration of electronics components. The configuration problem consists of configuring of the components such that they operate correctly individually and together.

The Experiment Control System

The configuration of the experiment is the responsibility of the Experiment Control System (ECS) [4]. The ECS needs:

- to know the information required to configure the experiment
- to be able to communicate with the different types of hardware and software,
- to verify whether the experiment has been configured properly and is in a correct state for taking data.

Configuring an experiment consists of settings the registers of electronics modules, downloading the right FPGA code [5], configuring software processes that are run on the data processing farm [6]. The configuration data consists of all the parameters and their values to be applied to hardware and software. The amount of configurable data depends on the type of the device. It can go from a few bytes to a few MB. It can consist of setting a few thousands of registers.

The links between the components have an influence on the configuration. There are millions of links. Each subsystem (subdetector) has its own connectivity topology. The connectivity must be described and is part of the configuration.

For instance, the TFC (Timing and Fast Control) [4] system which is responsible for the synchronization between detector components, will distribute the clock to the active electronics modules. The connectivity is required to determine to which electronics modules the clock should be forwarded, and which path to take when doing so. The DAQ (Data

Acquisition) system [4] is a Gigabit network with around one hundred of routers. The DHCP [7] and DNS [8] servers in this network need to be configured. Routing tables and DHCP config files have to be made. Certain electronics modules [9] contain lookup tables (similar to destination tables) which need to be made dynamically and downloaded. These tables depend on the connectivity.

Connectivity information also helps the ECS in designing an adaptive architecture in case of failures of modules. Using the connectivity, the ECS can derive the whole branch of devices to be isolated further to the failure of such a module and make the decision to go on or not with the data taking.

The granularity of the connectivity is important. For instance, there is a subsystem where tracking data path is very important. If data is not properly transferred, the subsystem group need to know not only which device fails but which component(s) nested in which motherboard is(are) faulty.

It is expected that the LHCb detector will take data over a period of around 10-15 years. This timescale puts important constraints on the configurability of the equipment. It is clear that on this timescale some equipment will fail and will need to be replaced. The huge amount of hardware in the experiment requires an automatic, reliable and reproducible system that can be used by the ECS to configure and manage the equipment.

Autonomics

HEP experiments become more and more complex in terms of technologies and number of items. Human errors, hardware or software failures are bound to happen. The ECS should be able to detect them and to react accordingly especially in the case where an error has not been predicted. Anticipating everything is not always possible. For instance, the data taking should go on if one sensor is badly configured or it should stop only if there are more than M sensors not correctly set. The value of M will certainly depend on the module type as a bad configuration of a sensor has not the same impact as a router wrongly configured. The experiment should then be adaptive. But also it requires knowing which modules are not properly configured and where they are located. Searching for a sensor among hundreds of thousands is a painful task if there is no mean to locate the faulty equipment.

Besides, budget and manpower are limited. It implies that the architecture should be as smart and self-managed as possible.

Autonomic computing takes its inspiration from the autonomic function of the human central nervous system. “Autonomic controls use motor neurons to send indirect messages to organs at a sub-conscious level. These messages regulate temperature, breathing, and heart rate without conscious thought. The implications for computing are immediately evident; a network of organized, "smart" computing components that give us what we need, when we need it, without a conscious mental or even physical effort” [10]. Thus autonomic tools are self-organising and dimensioning software.

Implementing autonomic tools is very useful and starts to be used in HEP experiments and LHCb is among them, especially in the Grid (data management). It allows reducing the human intervention (and consequently human errors). It enables the ECS to better configure and monitor the experiment. The ECS architecture is more robust and more reliable.

The ECS software architecture and its constraints

LHCb has an integrated ECS, that is, a unique and single control system for the whole experiment. Usually, HEP experiments are designed with two separate control systems or more (one control system per subdetector), one for the equipment which participates in the data taking (sensors, switches, routers, PCs, electronics board, etc.) and another one for the equipment which fuel the first group such as power supplies, High and Low Voltage, gas and cooling system. The second group consists of mainly commercial products and controlling them is slower (it is usually called the Slow Control system). The choice of an integrated ECS has pros and cons. It forces the experiment to follow certain guidelines but on the other hand the maintenance of the software is easier as they obey the same rules.

The ECS uses an industrial SCADA (Supervisory Control And Data Acquisition) system [11], called PVSS [12]. A SCADA system is a central system used to supervise a site or a process such as chemical, electrical processes which can execute logical processes without the master computer (which supervises everything). As described above it is of crucial importance for the ECS to be able to access information related to connectivity, configuration and history/inventory of devices. One of the tools created as part of this thesis is the CIC (Configuration Inventory Connectivity) DB, which is an Oracle database. In addition a set of smart and adaptive tools were created which allow interactions with the database.

The main constraints on the design of this database are:

- use of a single database which will contain information about configuration, connectivity and history/inventory and which should cater for all the subsystems[1]. It implies an unique and generic database schema;
- all the information needed by the ECS to configure the experiment should be stored in the CIC DB. Thus the information that the CIC DB contains should be complete.
- all the devices will be configured, monitored via PVSS. PVSS contains device drivers to allow communicating with the different types of equipment. Thus the CIC DB needs to have an interface for PVSS.

Objectives of the thesis

The main objective of this thesis is to provide the ECS with autonomic software tools which guarantee:

- a configuration of all the devices, whatever the type and whatever the settings needed, within one minute. It includes different types of configuration according to the physics studies. It should be thus flexible.
- a fault detection of devices with their location. It also aims at updating the configuration modules which are affected (such as routing tables of routers or look-up tables for instance).
- equipment management in a consistent and robust way as the detector has to be maintained for 10-15 years.

Contribution of the thesis

Methodology

[1] A subsystem is either a subdetector or the TFC system or the DAQ system. A subsystem is supervised by the ECS.

I have applied the following methodology to determine what information is needed by the ECS to configure the LHCb experiment properly and to design and implement the software architecture.

- Assimilating the LHCb environment with its different subsystems was one of the essential aspects. It implies understanding the jargon of the HEP world.
- First I had to identify the different groups of users. Then for each group I had to collect the requirements and use cases related to the project. I had to schedule at least one meeting with each subsystem (there are 8, TFC and DAQ included) to understand what kind of information they need to configure their subsystem. Sometimes it was not really clear since the electronics modules of a given subsystem were not yet fully operational. So there was a problem of time schedule. Also I got some contradictory use cases. In that case, I went to see my project leader and exposed him the problem.
  I contributed actively in the organisation of the CIC DB workshop [13]. I had prepared a questionnaire [14] to identify their expectations.
- I had designed the CIC DB schema using the list of use cases and the ERM [15]. I had collated the use cases and I had made several presentations of my work during Online group meetings and LHCb week meeting (the whole LHCb collaboration).
- I had implemented a set of autonomic tools which requires no SQL [16] typing and allows a consistent and robust manipulation of data stored in the CIC DB. I had also written the documentation of the code so that people can use it. I also helped the subdetector group and gave them some advices on how to use them and integrate them in their application. It was similar to a tutorial. I also wrote a C-template to insert the connectivity of a subsystem.
- Finally during the release of the different tools, I could verify if my tools and my table schema were corresponding to the needs, otherwise the users provided me with feedback. And I improved functionalities.

Software architecture

The software architecture I came up with is a 3-Tier Architecture composed of the three following layers:

- Database layer. It consists of the generic and relational CIC DB schema (with the indexes and the constraints) and a set of PL/SQL [17] routines. PL/SQL is very convenient to build complex SQL queries. A PL/SQL package **routingtable_pck** has been built to generate and update routing and destination tables. I have also implemented a set of PL/SQL functions to manage the status of the equipment.
- Object layer. It consists of a C-library (CIC_DB_lib) which provides a set of functions to manipulate the data (inserting, updating, deleting and querying) in a consistent manner. It uses OCI (Oracle Call Interface) [18] as DB interface. OCI and C are widely used in LHCb Online environment. Part of the functions embeds PL/SQL codes. A lot of checks have been integrated to preserve data integrity in case of human error such as mistyping or incoherent data such as a port of device connected twice. Two bindings have been implemented on top of the library, one in Python (also commonly used in LHCb group) using Boost [19] and one in PVSS (using GEH [20], Generic External Handler) so that the ECS can interact with the CIC DB. There are also two Perl scripts, one which creates the DHCP config file and another one which creates the DNS files. These Perl scripts can be embedded in C applications in combination with CIC_DB_lib, if needed.

- GUI (Graphical User Interface) layer. It covers all the PVSS panels which have been implemented by users to configure the devices and are using the PVSS CIC_DB_lib binding. There is also CDBVis, a Python tool based on the Python CIC_DB_lib binding to navigate through the CIC_DB and allow fault detection.

Fault detection and verification of correctness

Another essential aspect was to make sure that the data stored in the CIC DB was complete and correct. It has been achieved in the different layers of the architecture.

- Database constraints and triggers have been defined to ensure coherency in the update, insertion and deletion of information. For instance, a device cannot be inserted twice. PL/SQL programs are one of the essential components to ensure consistency of the information. If a port of a switch fails, the status of the link is updated automatically and paths going through this port are disabled.
- Check of input parameters given in the CIC_DB_lib, rolling back a transaction if something (such as inserting ports of a device without having inserting first the device) went wrong have been implemented to prevent from user mistyping or a bad usage of the tool.
- The ECS, via PVSS, can use the functions provided by the PVSS CIC_DB_lib extension to retrieve information about a device or connectivity between devices and compare with the current results. PVSS communicates with software and hardware via a system of commands and services (settings and reading back parameters). For instance, PVSS had noticed that a switch is down, because it does not respond. It uses the PVSS CIC_DB_lib to update the status of this switch. This triggers an update of the routing tables. Then PVSS loads the newly updated routing tables from the CIC DB, using PVSS CIC_DB_lib. And then it loads into the switches.
- CDBVis allows navigating through the CIC DB. It allows the user to verify that there are no mistakes and that the connectivity has been inserted properly. Here also a lot of checks have been built to ensure consistency and integrity.

Performance Issues

One of the requirements is to load all the information required to configure the experiment, from the CIC DB to PVSS in less than one minute.

A lot of tests have been carried out to tune the SQL queries and the PL/SQL programs. Most of programs have been reviewed by the CERN Database Support. With their help I also set up some stressing tests, where a lot of data are being inserted.

Performance improvement, which had required a lot of work, has been done to find paths between two devices. A requirement was it should not take more than a few seconds to find a path between two devices.

Another type of key tests is the simulation of concurrent accesses to the CIC DB. I implemented a C application which creates several processes which calls functions to query the paths. I wanted to make sure that the execution time fits still within the few seconds (requirement). This kind of tests is also interesting to check the behaviour of insert, delete and update functions.

Organization of this thesis

This thesis is organized as follows:

- The first chapter focuses on the LHCb experiment. It describes its architecture. It also presents the outlines of three important online subsystems: the TFC, DAQ and the ECS. The CIC DB is integrated into the ECS.
- The second chapter gives a brief introduction to the configuration of a HEP experiment. It presents the problems and what tasks need to be performed to configure the experiment. It gives an idea of the complexity of the system in terms of electronics modules and system topology. It also describes the software architecture which has been built to handle this complex task.
- The third chapter describes the outline of the software architecture. It explains how the software which interacts with the CIC DB has been built. It is a 3-tier architecture which consists of the GUI (Graphical User Interface), object and database layers. The database consists of two parts, the table schema and the PL/SQL programs. The core of the object layer is a C-library which manipulates the data. The GUI layer is composed of PVSS panels and CDBVis, a graphical editor. This architecture should guarantee reliability, consistency and adaptability.
- The fourth chapter presents the approach adopted to propose a reliable and adaptive architecture based on the CIC DB and autonomic tools. The users and requirements are identified. Requirements and the use cases have been collected through discussions and workshops. This step is essential to build an efficient system with a complete and consistent table schema and a set of autonomic tools.
- The fifth chapter explains how the CIC DB table schema was derived by applying the ERM (Entity Relationship Model) to the use cases found. Using the use cases, I have determined the entities and their attributes. The choice of the ERM is discussed and a brief reminder of the ERM is given. Also I describe another representation of the hierarchy based on prime numbers. This new model has been compared with the standard one (performance and limitation studies). I also show that the table schema is complete and the information required to configure the experiment can be stored in the CIC DB.
- The sixth chapter presents an application of the CIC DB to a computer network. The routing and destination tables and DNS and DHCP config files in the DAQ computer farm network are generated and updated automatically. The principles of the algorithm are presented. It is one of the key components of the architecture of autonomic tools. The algorithm has been implemented in PL/SQL and in C. The choice of the language is discussed. Using this algorithm I describe a method to automatically generate the DHCP config file. Also I explain how this algorithm has been extended to enable users to get the detailed paths between devices and between components of devices. I also explain how the DNS server can be configured using similar principles as for the DHCP config file. This set of tools contributes in building an autonomic control system.
- In the seventh chapter I describe the implementation of the autonomic tools to interact with the CIC DB. It is a focus on the object layer. The tools are based on a C-library which allows selecting, inserting, updating, and deleting information stored in the CIC DB. The C-library integrates check functions to ensure a consistent and reliable manipulation of data. It uses OCI to access the CIC DB. A binding to PVSS has been provided as devices must be configured from PVSS. I have also implemented another binding in Python used to build graphical editors.

- The eighth chapter describes the third layer of the architecture, namely the GUI layers. Different user interfaces have been implemented going from PVSS panels to CDBVis. CDBVis is a Python graphical editor which enables the users to navigate through the CIC DB to detect faulty equipment. Connectivity can also be inserted with this tools
- The ninth chapter presents the different tests carried out to validate the table schema and the algorithms related to paths. I have exercised the algorithm against different types of topologies to test its robustness. I also have tested that the routing tables generated could be programmed in real switches and routers. The DHCP config file generated by our program has been put in a DHCP server to ensure compatibility. Same remark for the DNS.
- The tenth chapter presents the different tests carried out to validate these tools, mainly in the form of their integration into applications such as the TFC, and subdetectors. Several groups are using them for different purposes such as configuring the network, computing configuration parameters values (gain, voltages), etc.
- The eleventh chapter describes how other large experiments are configured. I present the ATLAS thin gas chambers, the VLT (Very Large Telescopes) [21] and ALMA (Atacama Large Millimeter Array) [22] projects. The two latter projects are related to telescopes. The aim of this chapter is to show that our approach is generic enough to be applied in other fields than LHCb.
- Finally I give some conclusions and suggestions for further extensions.

References

[1] CERN website, http://public.Ib.cern.ch/public/.

[2] LHC website, http://lhc.Ib.cern.ch/lhc/.

[3] J. H. Christenson, J. W. Cronin, V. L. Fitch, and R. Turlay, Evidence for the 2π Decay of the K_2^0 Meson, 10 July 1964. In Physical Review Letters, Volume 13, number 4. p 138-140.

[4] LHCb Collaboration, LHCb Online System Data Acquisition & Experiment Control, Technical Design Report. CERN/LHCC 2001-040, LHCb TDR 7 December 19th 2001.

[5] Shebli Anvar, Olivier Gachelin, Pierre Kestener, Herve Le Provost and Irakli Mandjavidze, FPGA-Based System-on-Chip Designs for Real-Time Applications in Particle Physics. IEEE Transactions on Nuclear Science, June 2006, Vol. 53, number 3, Part I of three parts. 14th Conference on real time (RT 2005) Stockholm, Sweden, June 4-10, 2005. p667-670.

[6] E. van Herwijnen, L. Abadie, A. Barczyk, B. Damodaran, B. Gaidioz, C. Gaspar, R. Jacobson, B. Jost, N. Neufeld, Control and monitoring of on-line trigger algorithms using a SCADA system. Proceedings of the CHEP 06 (Computing in High Energy and Nuclear Physics) Conference, Mumbai, India, 13-17 February, 2006.

[7] Dynamic Host Configuration Protocol, RFC 2131, March 1997, http://www.ietf.org/rfc/rfc2131.txt.

[8] Domain Name System Protocol, RFC 1035, November 1987, ftp://ftp.is.co.za/rfc/rfc1035.txt.

[9] LHCb Collaboration, LHCb Trigger System Technical Design Report. CERN/LHCC-2003-031, LHCb TDR 10, September 2003.

[10] IBM research: Autonomic computing
http://www.research.ibm.com/autonomic/index.html

[11] A.Daneels, W.Salter, What is SCADA? October 8th, 1999. In the Proceedings of the International Conference on Accelerator and Large Experiment Physics Control Systems, 1999, Trieste, Italy.

[12] ETM, PVSS II, http://www.pvss.com/index_e.asp?id=2&sb1=6.

[13] LHCb ConfDB workshop, http://indico.cern.ch/conferenceDisplay.py?confId=a055409.

[14] ConfDB questionnaire
http://lhcb-online.web.cern.ch/lhcb-online/configurationdb/Documents/confDB/Configuration%20Database%20Questionnaire.doc.

[15] DATABASE SYSTEMS, a practical approach to design, implementation, and management written by Thomas Connolly and Carolyn Begg, 2002. Third Edition. ADDISON-WESLEY. ISBN 0−201−70857−4. 1236 p.

[16] SQL Structured Query Language SQL-92 Standard, July 1992
http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[17] Steven Feuerstein, Bill Pribyl, Oracle PL/SQL Third Edition, USA: O'REILLY, September 2002. 0-596-00381-1. 1018 p.

[18] ORACLE, Oracle Call Interface, Programmer's Guide, 10g release 2. ORACLE PRESS, OSBORNE. B14250-02, November, 2005. 1258 p.

[19]Boost Python, http://www.boost.org/libs/python/doc/

[20] GEH, Generic External Handler

http://itcobe.web.cern.ch/itcobe/Projects/Framework/Download/Components/GenericExternalHandler/welcome.html
[21] VLT project,
http://www.eso.org/projects/vlt/sw-dev/wwwdoc/APR2004/dockit.html
[22] ALMA project,
http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ConfigurationDatabase.pdf

Chapter 1 Overview of the LHCb experiment

This chapter gives an introduction to the LHCb experiment. It is one of the four experiments at the LHC. First we explain the purpose of the LHCb experiment. Secondly we describe the structure of the LHCb experiment by presenting the different subdetectors, the Online and Offline systems. Thirdly, we put the emphasis on the ECS architecture and show how the use of autonomics tools can help with the configuration of the experiment.

1.1 Purpose of HEP experiments

HEP is the study of elementary particles (photons, electrons, quarks etc.) and the interactions (electric and magnetic forces unified with the weak and strong forces in the Standard Model, gravity) between them. In HEP experiments, nature is compared to theory in a reproducible and measurable way. A detector enables physicists to carry out their measurements, i.e. the experiment. HEP experiments try to answer to fundamental questions about nature:

- Where does the mass come from?
- Are particles elementary or can they be split into smaller components?
- What is the origin of the Universe?

HEP experiments use powerful machines, such as the LHC at CERN, to accelerate elementary particles. Particles traveling near the speed of light and in opposite directions can collide and create other particles. Then depending on the results of the interaction, physicists can analyze the properties of the colliding and the created particles, frequently very short lived and their interactions.

HEP detectors register the collisions between these particles. For instance, the LHC replicates the conditions when the Universe was only a hundredth of a billionth of a second old. An event is a collision between particles. In most cases, the collision results just in the deviation of trajectories (elastic collisions).
HEP experiments involve many institutes from different places in the world which design and build pieces of the experiments (hardware and software). It is important to make sure that the integration is properly done.

1.2 The characteristics of the LHCb experiment

The LHCb detector is shown in Figure 15. LHCb is one of the four experiments at LHC. Two beams of protons will be accelerated to the highest energy (14 TeV) ever achieved in a laboratory to allow the creation of new particles.

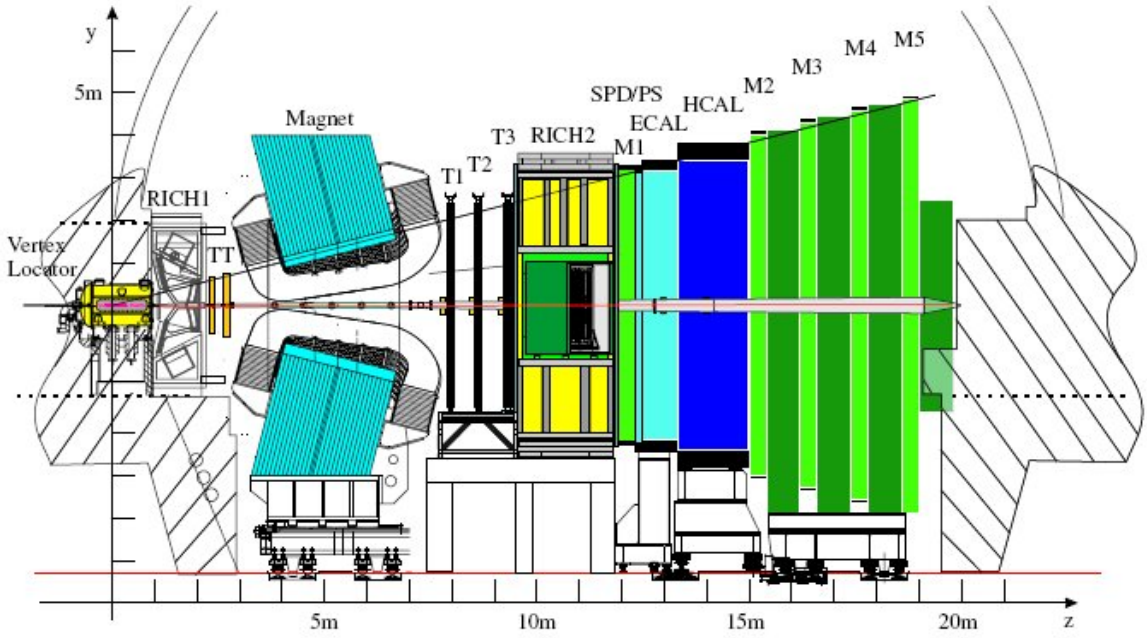


Figure 15. Side view of the LHCb detector.

1.2.1 Physics objectives

The laws of physics are founded on nature's symmetries. Symmetry is an aspect of an object that remains unchanged when something is done to it. For example, a chair remains a chair even when it is rotated upside down. Many of the dynamical laws of physics are a direct consequence of underlying symmetries. For example, the law of conservation of energy turns out to be the consequence of time translation symmetry – the fact that if you do an experiment today, it will have the same result as yesterday.

There are three important discrete symmetries in nature:

- Invariance under charge conjugation (C). This means changing the sign of the charge of a particle: in fact this symmetry changes a particle into its antiparticle.
- Invariance under parity (P). This means the sign of the coordinates are changed. The laws of physics are the same when we observe nature from a mirror.
- Invariance under time reversal (T)

It turns out that although violations of these symmetries are found, the product of the three, CPT invariance is preserved. CP violation was first detected in 1962 [1]. Kaons are more likely to decay in positrons (e+), on the left than in electrons (e-) on the right as illustrated in Figure 16.

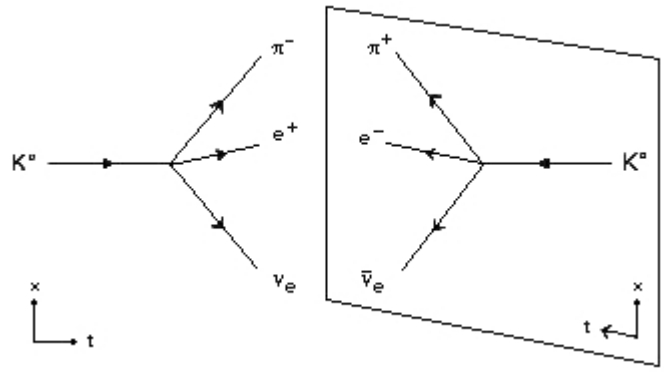


Figure 16. Example of CP violation: Kaon decays.

The origin of CP violation is still one of the outstanding mysteries in particle physics. Although the standard model can account for the observed CP violation in the Kaon system the observed effects could also be accounted for by new physics. With the high energy of the LHC, the b-production cross section is very large, so that a great accumulation of b-events can be collected, allowing precision measurements of the relevant observables. Furthermore, a number of states not accessible to the b-factories, such as Bs and Bc can be studied. The Standard Model makes precise predictions for the amount of CP violation that should be observed in a large number of B meson decay modes.

CP violation also plays an important role in cosmology. There is an apparent excess of matter over antimatter in the Universe and the amount of CP violation predicted by the Standard Model is not sufficient to explain this excess.

The B meson system is therefore an attractive place to search for CP violation and to search for a hint of new physics.

The LHCb detector, unlike other LHC detectors, does not provide a full 4 π coverage (i.e. the whole space). This geometry was chosen because from computer simulations it is known that at high energies both the b and $\bar{b}$ (antiparticle) hadrons are produced in the same forward cone. (There is also a cone pointing backward but for reasons of cost, the detector only looks at the forward cone). The detector is a single-arm spectrometer with a forward angular coverage or acceptance from 10 to 300mrad. It is 20m long and 10 m wide.

Similarly, the average decay length for B mesons can be calculated to be around 1.0 cm (see Figure 17).

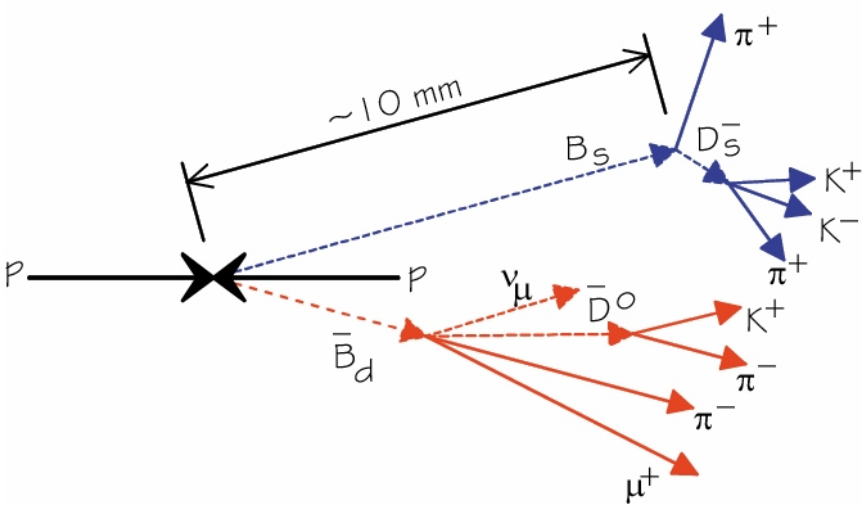


Figure 17. A typical p-p collision producing 2 types of B mesons.

1.2.2 Subdetectors

The LHCb detector is composed of several subdetectors that each has a specific task. During a normal physics data taking run they will all be up and running; however during debugging runs they can be operated independently:

- **Vertex Locator** (VELO) [2]. The purpose of the VELO is to determine the position of the collision and the secondary vertices that characterize the events that could display CP violation.
- **Pile-Up System** (PUS) [2] is located upstream from the VELO. Its task is to identify and reject bunch crossings where multiple p-p collisions occurred at the same time. This is because analysing such as collisions is more difficult. It is not represented in Figure 15.
- **The RICH** (Ring Imaging Cherenkov) [3] subdetector is divided into 2 parts, the **RICH1** before the magnet and the **RICH2** after the magnet. The **RICH** subdetectors will allow charged pion/kaon separation.
- **The Trackers** consist of 3 stations (T1, T2 and T3) and a Trigger Tracker (TT). T1, T2 and T3 are located between the magnet and RICH2. The **Trigger Tracker** is between the RICH1 and the magnet. The main objective of the tracking system is to provide precise measurements of momentum of charged particles. Each tracking station (T1, T2 and T3) has an **Inner Tracker** (IT) and an **Outer Tracker** (OT) [4]. The Inner Tracker and Trigger Tracker (TT) constitute the **Silicon Tracker** (ST) [5]. The OT uses aluminium frames and straw tubes.
- The **Calorimeters** [6] will permit identification of electrons, hadrons and photons. They will also provide energy and position measurement. There are 4 subdetectors, **PreShower** (PS), **Scintillator Pad Detector** (SPD), **ECAL** (electron calorimeter) and **HCAL** (hadron calorimeter).
- The **Muon** [7] detector is composed of 5 stations. Muons appear in the final states of the decays. The objective of the Muon System is to allow muon identification which is required by the trigger.

1.2.3 Online system

In addition to the subdetectors and the front-end electronics, the entire system related to the operation of the detector is referred to as the "Online System". It is a large real time system, often with strong requirements on the response time. Globally, it can be viewed as consisting of four subsystems with respect to their tasks and requirements, **TRIGGER**, **TFC**, **DAQ** and **ECS** (see Figure 18).

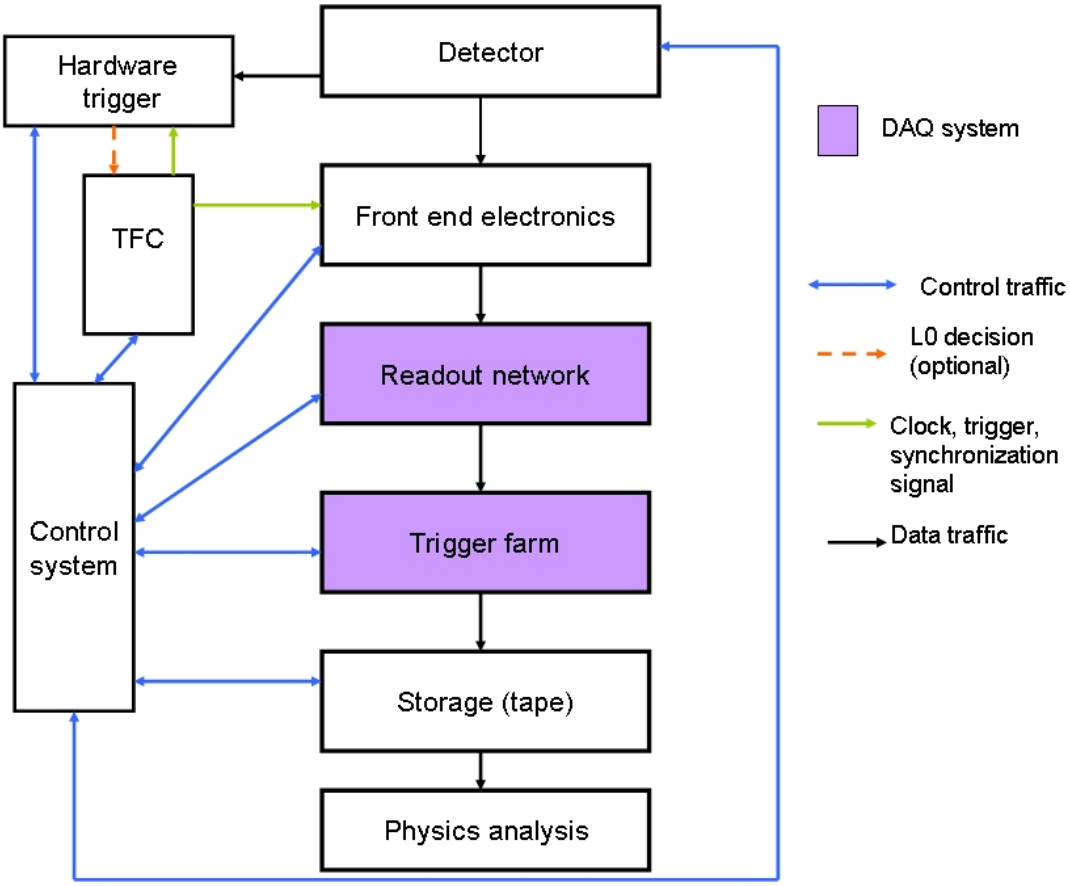


Figure 18. From the detector to tape.

1.2.3.1 The Trigger

At LHCb, there are two levels of trigger, the Level 0 (L0) trigger [8] and the High Level Trigger [9] (HLT).

For this thesis it is important to note that the timescale mentioned in the next two subsections is independent of the timescale on which the detector should be configured. The configuration may take seconds or even minutes.

1.2.3.1.1 The L0 Trigger

After a collision each subdetector will measure specific parameters. The probability that two protons collide is low, roughly 30% of the bunch crossings[2]. The probability that the collision produces B mesons is even lower, about 1 out of 160 events.

[2] The particle beams are formed in "bunches" of a few centimeters in length and the size of a hair in cross section. So a bunch crossing corresponds to the collisions between two beams of particles.

The trigger reduces this rate to a manageable amount, both in terms of readout speed as well as in terms of the required storage space.

The L0 trigger is implemented in the hardware and is based on the output of three subsystems, the Pile-Up, the calorimeters and the Muon system. It reduces the event rate from 40 MHZ to 1 MHz. It will select the most interesting events containing B decays with a high transverse momentum and reject less interesting events or events with multiple (>2) vertices . The decision to keep or reject an event has to be taken within 4 microseconds. During that time, roughly equivalent to 160 bunch crossings, the L0 front electronics will store the events in a pipeline memory.

1.2.3.1.2 The HLT

The HLT (High Level Trigger) is based on software and run on the event filter farm. It reduces the event rate to a few kHz.

The HLT Trigger algorithm running on farm PCs processes full events. A full event, i.e. the collection of all fragments of an event sent by the TELL1 boards at the same time, is treated by one single farm node. Data come at 1MHz and is processed in the Event-Filter-Farm (EFF) farm composed of roughly 2000 PCs. The HLT algorithm must take at most 2ms per event. The HLT algorithms need to be configured according to the running mode as the selection of the events depends on the physics studies.

1.2.3.2 The DAQ

1.2.3.2.1 Architecture

The DAQ is responsible for sending the selected data to permanent storage. It uses a Gigabit Ethernet based on IP protocol. There are two separate networks (different switches, different links), the data network and controls network. Each of them has their own equipment. The data network is used to carry the events' data coming from the detector. The controls network is used to configure and monitor the equipment.

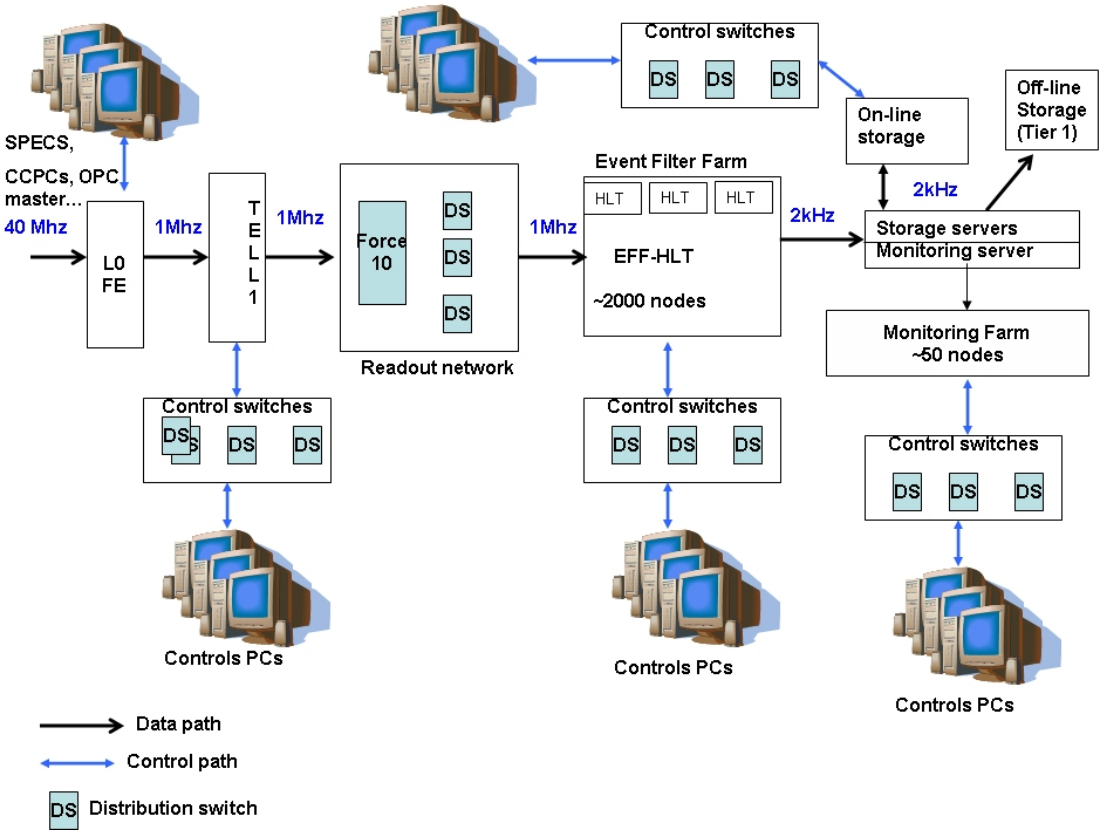


Figure 19. Overview of the DAQ system which includes the controls and data paths.

As shown in Figure 19, in the data network, the TELL1 boards are sources and the farm nodes (in the Event Filter Farm) are destinations.

1.2.3.2.2 The TELL1 boards

The TELL1 boards (see Figure 20) receive data only if the L0 Trigger accepts the events. They clean and digitize the signals. They process and format the fragments of several events according to the MEP (Multi-Event Packet) protocol [10]. This protocol consists of grouping several (N) fragments of events in one packet. N is called the packing factor and can be changed during a run. This value is sent by the readout supervisor (see section 1.2.3.3) to the TELL1 boards. The MEP is integrated in IP packets as shown in Figure 21. TELL1 boards have 5 Gigabit interfaces (2 data interfaces and one controls interface).

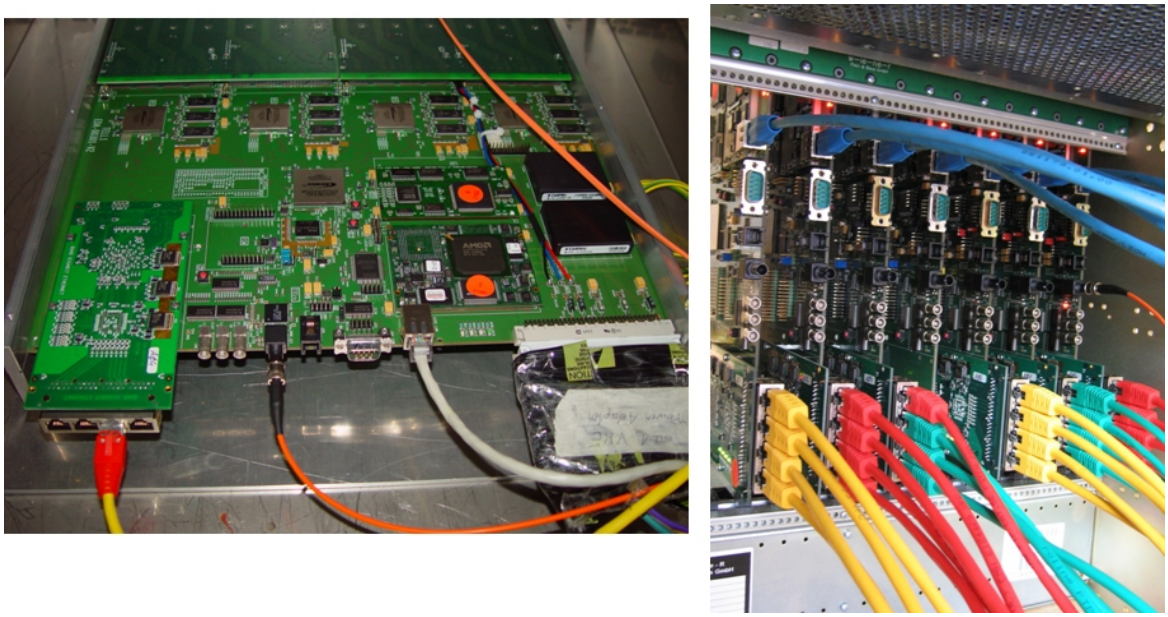


Figure 20. One TELL1 board on the left and 6 TELL1 boards on the right.

Figure 21. Overview of the MEP embedding.

1.2.3.2.3 The farms

There will be two types of farms (see Figure 19).

The **event filter farm** (EFF) runs HLT algorithms. Each farm node receives MEP packets so that full events can be reconstituted (the number of events built up will depend on the packing factor which is included in MEP packets).

The **monitoring farm** will allow subdetectors to run special analysis jobs to determine whether a subdetector is running correctly. A partial reconstruction will allow the monitoring of the physics data quality. These analysis tasks produce histograms that will be displayed by the control system. Part of the histograms will be produced using the events rejected by the EFF.
Sources and destinations are connected by switches and routers.

1.2.3.2.4 Permanent storage

If the event has been accepted by the HLT, it goes to the CERN mass storage system called CASTOR (which is hosted by CERN's IT department) [11]. However the storage network in the pit is capable of storing between 5 and 10 days (depending on the amount of data which need to be written) of events in case of problems with the storage on Castor. It is foreseen to store 1 PB of data per year.

1.2.3.3 The TFC

1.2.3.3.1 Architecture

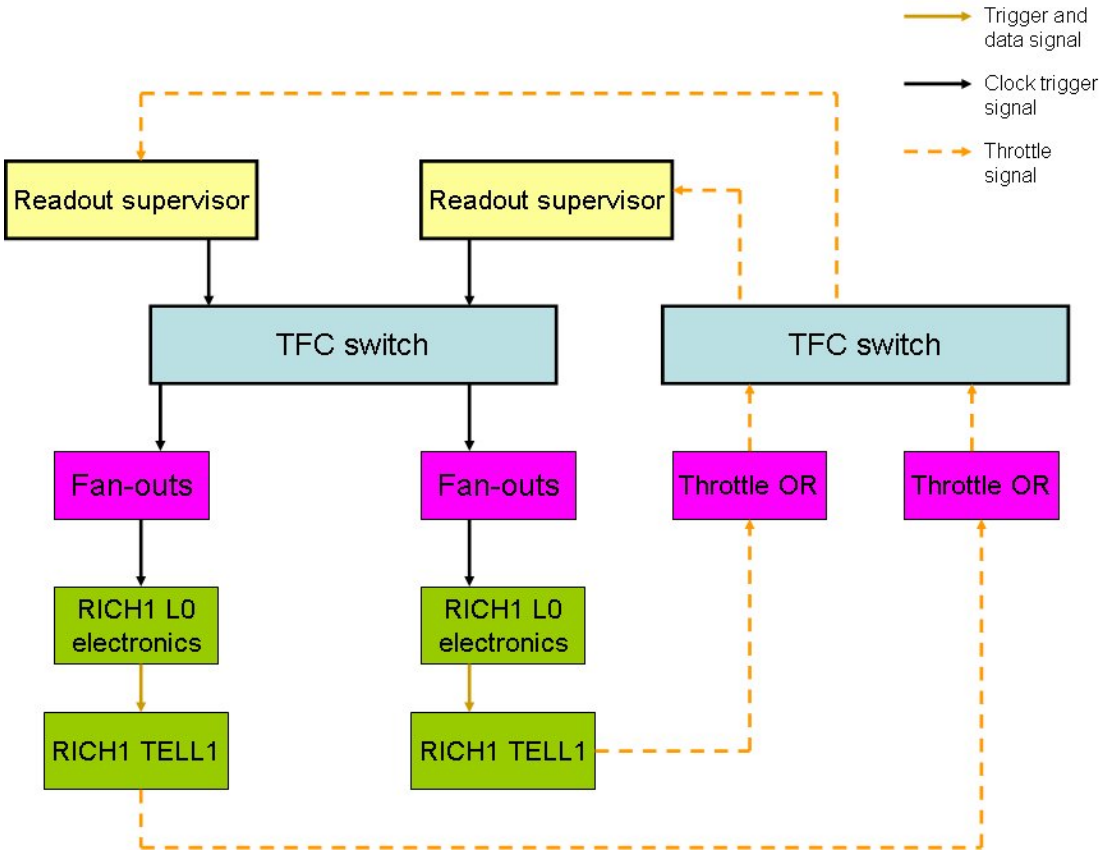


Figure 22. Simplified schema of the TFC connectivity.

The TFC system will receive the LHC clock and send it to all the L0 electronics and TELL1 boards for synchronization. It will also forward the L0 trigger decision to the L0 boards.

Referring to Figure 22, the TFC system consists of several kinds of equipment.

The **readout supervisors** are the masters of the TFC system. They fulfil different tasks:

- clock distribution from the LHC clock,
- forwarding and processing the L0 decision to the L0 electronics, TELL1 boards,
- destination assignment of the event,
- buffer overflow control,
- auto generating test triggers (calibration, monitoring and debug),
- sending synchronous reset commands
- sending a fragment of an event (specific information about a run such as the run number, the event ID, and the orbit signal, etc.)

The registers of these devices must be configured. There are also FPGA codes to download.

The **Fan-outs** or passive switches split the signal. These switches cannot be configured.

The **TFC switch** (see Figure 22) has 16 inputs and 16 outputs. Ports can be disabled or enabled. The TELL1 boards of a given subsystem are connected via several fan-outs to one or two outputs of the TFC switch. None of the outputs can be connected to two different subsystems because the partition (ensemble of subdetectors selected to

take data, see 1.2.5 for more details) will be based on the connectivity of this switch. The internal connectivity of this switch must be configured. It is derived from the selected partition. It will be explained in more details in Chapter 2.
The **throttle switches** are in blue. They are responsible for sending a fast alarm to the readout supervisors in case of buffer overflow in the L0 or TELL1 boards. The internal connectivity of this switch must also be configured. There is a correspondence between the internal connectivity of the throttle switch and the TFC switch. Indeed the output port numbers used in the TFC switch must correspond to the input port numbers of the throttle switch. The input port numbers used in the TFC switch must correspond to the output ports of the Throttle switch. This concept is shown in Figure 23.

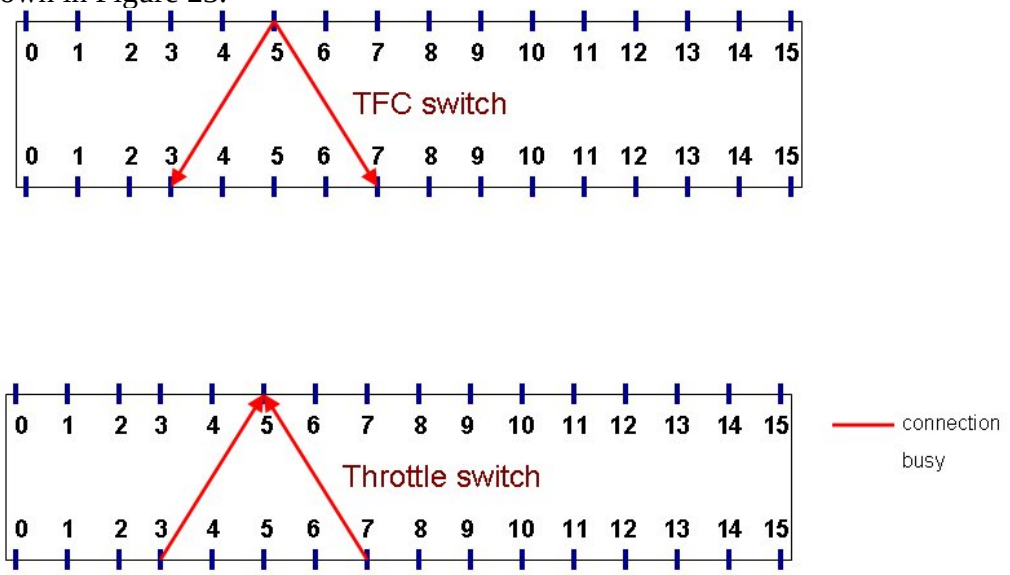


Figure 23. Example of internal connectivity. Inputs of the TFC switch are on the top whereas the inputs of the Throttle switch are on the bottom.

Thanks to this mapping, we can know which subsystem(s) has (have) triggered an alarm.

1.2.3.3.2 Interaction between the TFC and the DAQ

The TELL1 boards need to specify the IP address of the destination farm node when building their MEP packets. They get this information from the readout supervisor who knows which farm nodes are free. Indeed when a farm node is free, it sends a request for events to the readout supervisor via the Force Ten router. Then based on the dynamic destination assignment algorithm [12], it selects one farm node among the free ones.

1.2.3.4 The ECS

The ECS is responsible for the control and monitoring of the detector and the data. It is based on PVSS, a SCADA system.
See 1.3 The ECS.

1.2.4 Offline

The Offline system consists of all the physics software used to analyze and reconstruct tracks based on the data stored during data taking.

It uses farms of PCs for simulation, reconstruction and analysis.. Technologies based on Grid [13] such as DIRAC [14] allow data access to universities outside CERN using computing resources from all around the world.

To test and validate the algorithms, the Offline System organizes Data Challenges. During that time, events are produced using Monte Carlo programs for simulating the behavior of the detector.

1.2.5 Operating the detector

An **activity** corresponds to a configuration of the detector. It will depend on the type of data being taken. Examples of types of data taking activities are physics (the normal situation), alignment (to determine the position of the detector with respect to the beam), calibration, cosmics, testbeam etc. One essential task is the alignment of a subdetector with the beam. Alignment will determine the quality of measurement. It should be possible to control and configure a subdetector independently of the others to allow a very precise alignment. A subdetector will be configured differently if it is used for alignment or for physics. The number of parameters to load into devices depends on the type of running mode. Usually there are millions of parameters to load into the equipment.

The detector of a HEP experiment consists of several subdetectors which have to perform a specific task (e.g. alignment mentioned above, determination of track parameters, etc.).

Subdetectors are designed and built by several institutes. The equipment will be integrated and commissioned. Consequently it should be possible to test part of a detector. For these reasons, HEP experiments define the concept of **partitioning**. A partition is a set of subdetectors which can run independently and concurrently. It is possible to define one or several partitions at the same time depending on the architecture of the experiment.

A **run** corresponds to the period of physics data collection. The detector must be configured to allow a run. A run ends when the DAQ stops taking data. However the detector is still powered on. The partition (i.e. which subsystem will take data) and the activity (how devices will be configured) must be specified.
When the detector becomes operational, i.e. starts to take data, operators will be on shift to make sure that everything is running according to plan. However failures or unforeseen behavior of the equipment are bound to happen. Automatic programs are not sufficient by themselves to ensure the control of the detector. Configuring, monitoring and control of the detector will be handled by the control system. The operators will inspect histograms produced by the software to check if the detector behaves as expected.

The operator can start and stop the detector. If an error occurs that cannot be fixed, the operators can call an expert to solve the problem. Tools are required to allow a good visualization of the essential information and also provide all the information needed to operate the detector in case of problems. If an electronic board fails, all the devices which are affected by this failure must be known for subsequent analysis.

People on shift are trained to know the basic rules so that they can intervene when a simple problem occurs.

1.2.6 Equipment management

Usually HEP experiments run over many years, around 10 years. Part of the equipment is exposed to radiation. They are more likely to break down. They will be replaced. Thus it is necessary to manage equipment, i.e. at any time the number of spares by types or the location of any devices should be known.
Also all the software parts should be maintained and sometimes should be updated according to the technology evolution.

The longevity of the experiment also poses a stringent constraint on the robustness of the software and the underlying tools, e.g. Oracle.
To permit the detector to run over years, inventory and history mechanisms are required.
How many spares of a given device type are left? Where is a given module located? How often does a given device fail? All these kinds of questions are very important to maintain the detector.

1.3 The ECS

The ECS is responsible for supervising the experiment.

1.3.1 Control system architecture

Figure 24 depicts the architecture of the ECS. It consists of different components:

- PVSS which is the SCADA system used to control and configure the LHCb experiment. (See 1.3.2.1 PVSS for more details). It is the central part which communicates with all the other components of the ECS using LHCb frameworks (specific set of tools developed by LHCb and integrated in PVSS);
- DIM (Distributed Interface Management) system [15] which permits to send commands and receives services between PVSS and the modules (see 1.3.2.2)
- FSM (Finite State Machines) [16] which permits to model the behavior of the experiment using states and actions (see 1.3.2.3 for details);
- The CIC DB, an Oracle DB, which contain the information necessary to configure the experiment (see Chapter 3 for the software architecture);
- The Conditions DB (Cond. DB) [17], an Oracle DB, which contain snapshots of the time-dependent parameters used for physics computing such as environmental parameters;
- The PVSS archiving DB, an Oracle DB which contains snapshots of some monitoring values used for subsequent analyses.

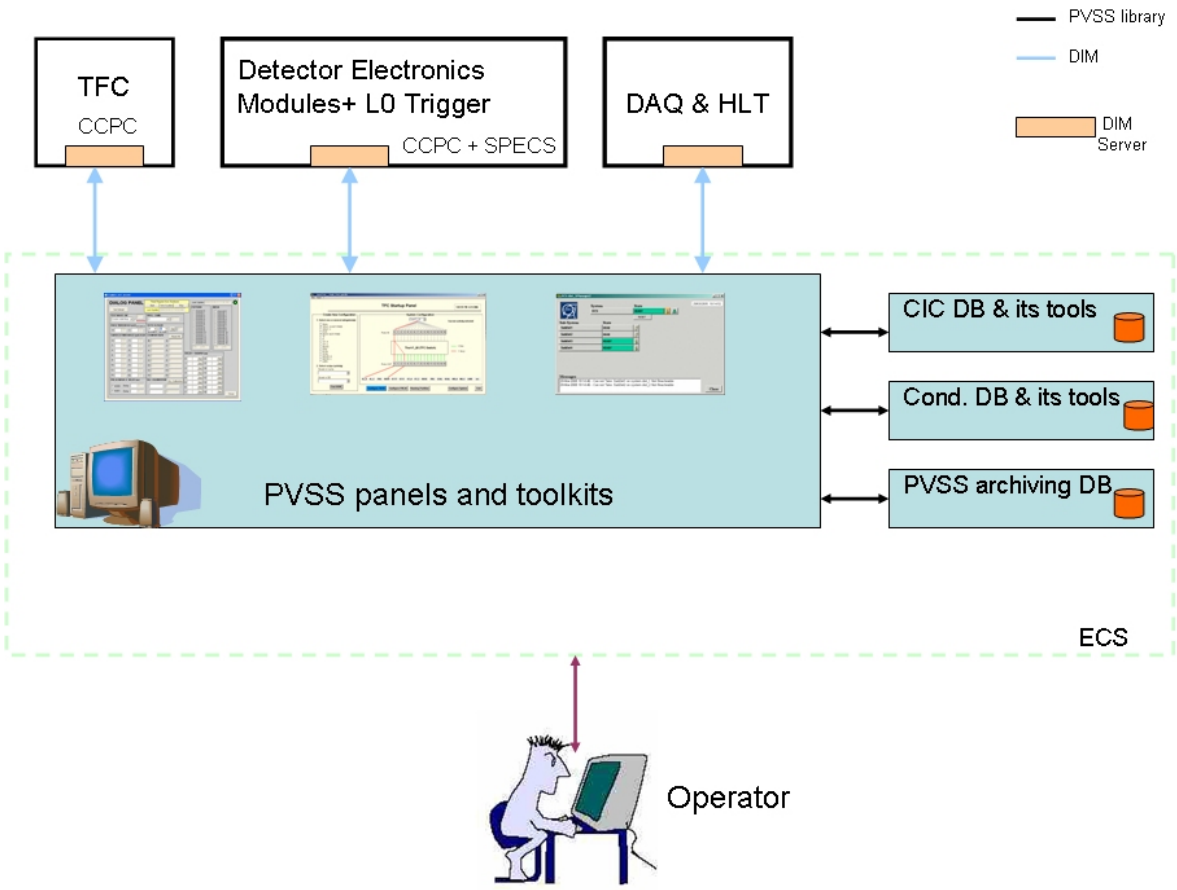


Figure 24. The ECS software architecture.

The ECS consists of a hierarchy of a controls PCs where PVSS and LHCb specific framework are installed. The characteristics of this hierarchy are described in section 1.3.2.3.

Each controls PC is responsible for a part of the equipment.

At start up of data taking the controls PCs will then send requests to the CIC DB to determine what and how electronics modules should be configured. The CIC DB will provide the controls PCs with the necessary information. Then the controls PCs will configure the equipment they supervise.

Data taking will start when all the equipment has been configured. To allow tracking reconstruction, some parameters (temperature, pressure for instance) must be measured and saved in the conditions DB.

The online acquired data used to monitor the detector is saved in the PVSS archive database. Finally, at the end of a run, it is also possible to save configuration settings into the CIC DB.

1.3.2 Controls software

1.3.2.1 PVSS

To build a control system with PVSS we have to model the detector components in the hierarchy required by the ECS. The LHCb experiment is composed of many devices. It can be modelled as a tree of PVSS “datapoints”.

Each device is associated to a data point (DP), an instance of a data point type (DPT). Data point types are similar to object oriented classes with their attributes. For instance, for the type (class) of devices “TELL1” there will be a corresponding DPT. VELO_TELL1_22 and MUON_TELL1_12 (the instances) will have DPs of DPT TELL1. They will have the same structure but with different values. All the configurable and controllable electronics modules will be modelled using DPTs and their configurable parameters will be stored inside DPs.

PVSS provides support for setting and reading values to/from hardware. Communication between the commercial hardware (High/Low voltage for instance) and PVSS takes place via the OPC protocol [18] provided by manufacturers such as CAEN [19], WIENER [20], etc. (not represented in Figure 24)

For home-grown equipment such as L0 electronics, TELL1 boards, the communication is done via DIM (see Figure 24). Then a DIM server receives and sends information to the real hardware via SPECS [21] or CCPC [22], protocols developed by LHCb. SPECS tends to be used for equipment exposed to radiation, CCPC for TELL1 boards.

Panels and scripts are implemented to control and configure devices. Figure 25 shows an example of a PVSS panel which displays and permits to change important hardware parameters.

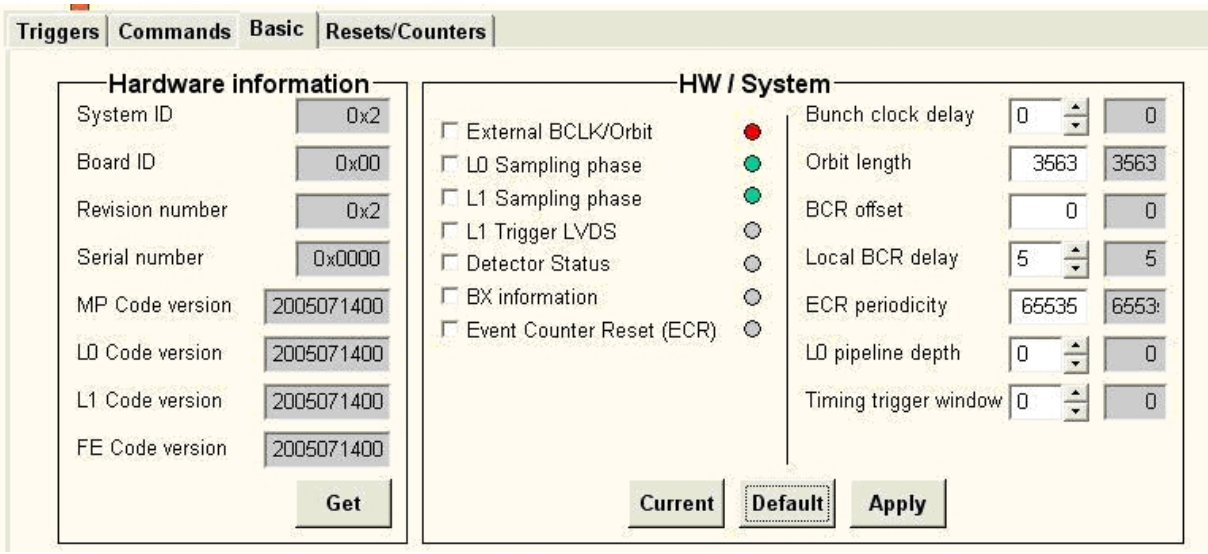


Figure 25. Example of a PVSS panel.

1.3.2.2 DIM

DIM is a communication system for distributed / mixed environments, developed at CERN. It provides a network transparent inter-process communication layer. At LHCb, it will be used to ensure communication between modules and PVSS. Thus configuration information will be transmitted to the modules via DIM. DIM servers need to be implemented to translate the information between PVSS and the hardware equipment. Then, CCPC or SPECS are used to transfer the information between the DIM server and the hardware equipment. In the case of the DAQ, the communication between switches, farm PCs are direct with the DIM servers.

1.3.2.3 FSM

The LHCb experiment will be modelled as a tree as shown in Figure 26. The leaves of the tree correspond to the experimental equipment. The other nodes correspond to logical objects used to model the detector. The parent supervises its children.

A Finite State Machine (FSM) mechanism has been integrated with PVSS using SMI++ [23]. The control of the experiment using states, commands, transitions and actions is realised using this FSM.

Each node has 5 possible states {NOT_READY, READY, RUNNING, ERROR, UNKNOWN} as shown in Figure 27. The FSM framework provides panels to enable the user to define the different transitions using conditions such as *if ALL the children are in state READY, go to state READY.* Actions, which need to be triggered after or before a change of state or during a transition, can be implemented using scripts.

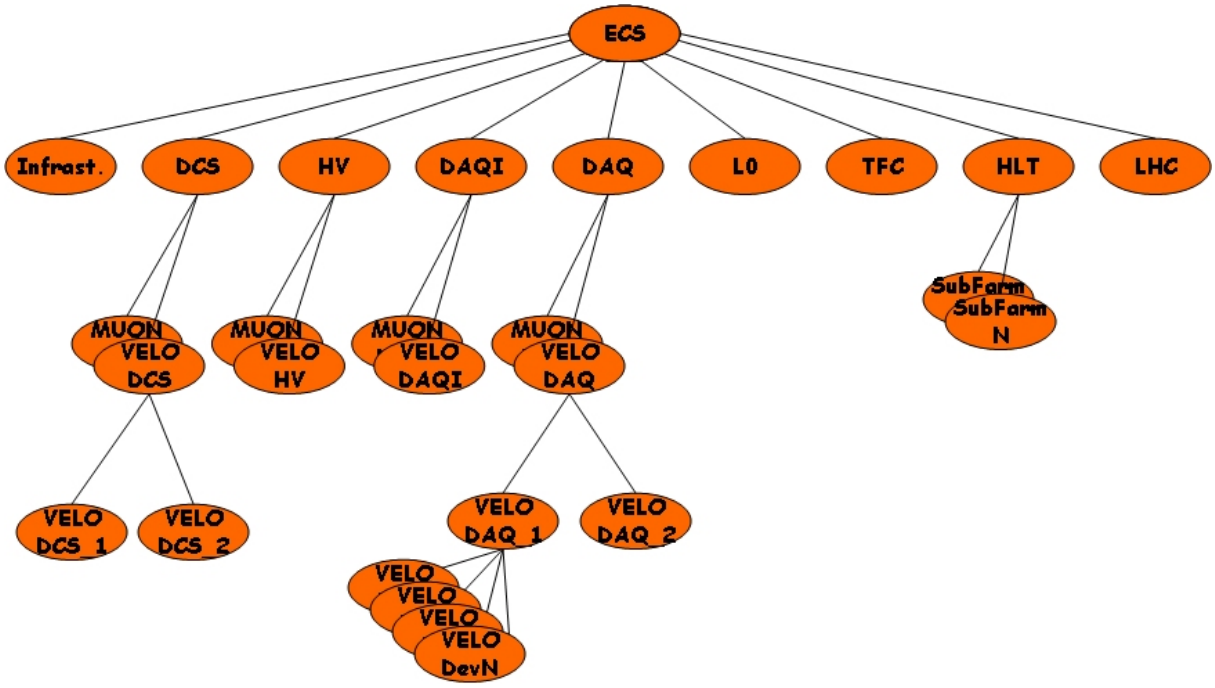


Figure 26. The LHCb experiment modeled as a tree.

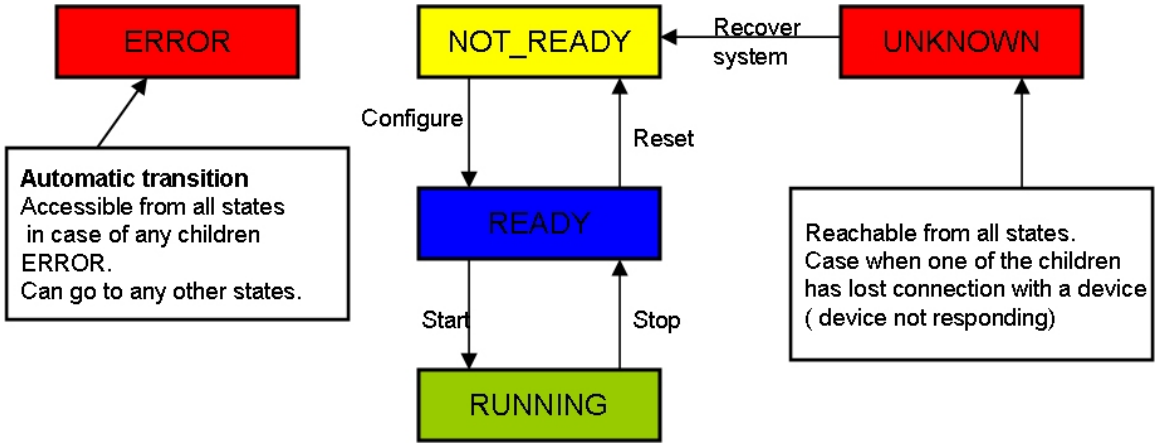


Figure 27. States and transitions of the detector.

Each subsystem has to design their own sub-tree following guidelines[24]. They have to define the transitions, i.e. which operations to perform to go from a state to another one. They also have to handle errors, for instance, what to do if one of the children is not in state READY.

1.3.2.4 Modeling the behavior and states of a device with FSM

The behavior and states of each module should be represented. When a configuration is applied to one or a set of devices, it should be possible for the shift operator to know the devices which should have been configured, the ones which have been properly configured and the ones which not. Also if a power supply, PS_45 for instance, fails, the information “power supply PS_45 fails” should be propagated to the devices affected by this failure. And these latter should change their behavior and state and react.
So for each module type, a set of states and transitions associated with actions should be defined by the designer using the FSM guidelines and toolkit.

1.3.3 Use of the CIC DB and its autonomic tools

The CIC DB has been implemented using Oracle. It will contain information to permit the configuration of the experiment. It covers three types of data:

- Configuration information which corresponds to register values of devices, FPGA code to download, etc. A configuration is identified by a list of devices and a collection of (parameter, value)-pairs per device of the list.
- Connectivity information consists of representing links between devices. It is used to configure the DAQ network (routing tables for instance) to handle partitioning for the TFC, in some algorithms to compute parameter values, and to check that the data path is properly set.
- Inventory and history information to manage and trace the equipment.

Associated with the database, we have developed a set of autonomic tools which automates the creation and updates of routing tables and destination tables. They also permit a non-DB expert to safely manipulate the data by reducing the human intervention and by implementing check functions.

1.3.4 Configuring the detector

At start up, all the subsystems will have the state NOT_READY as shown Figure 28.

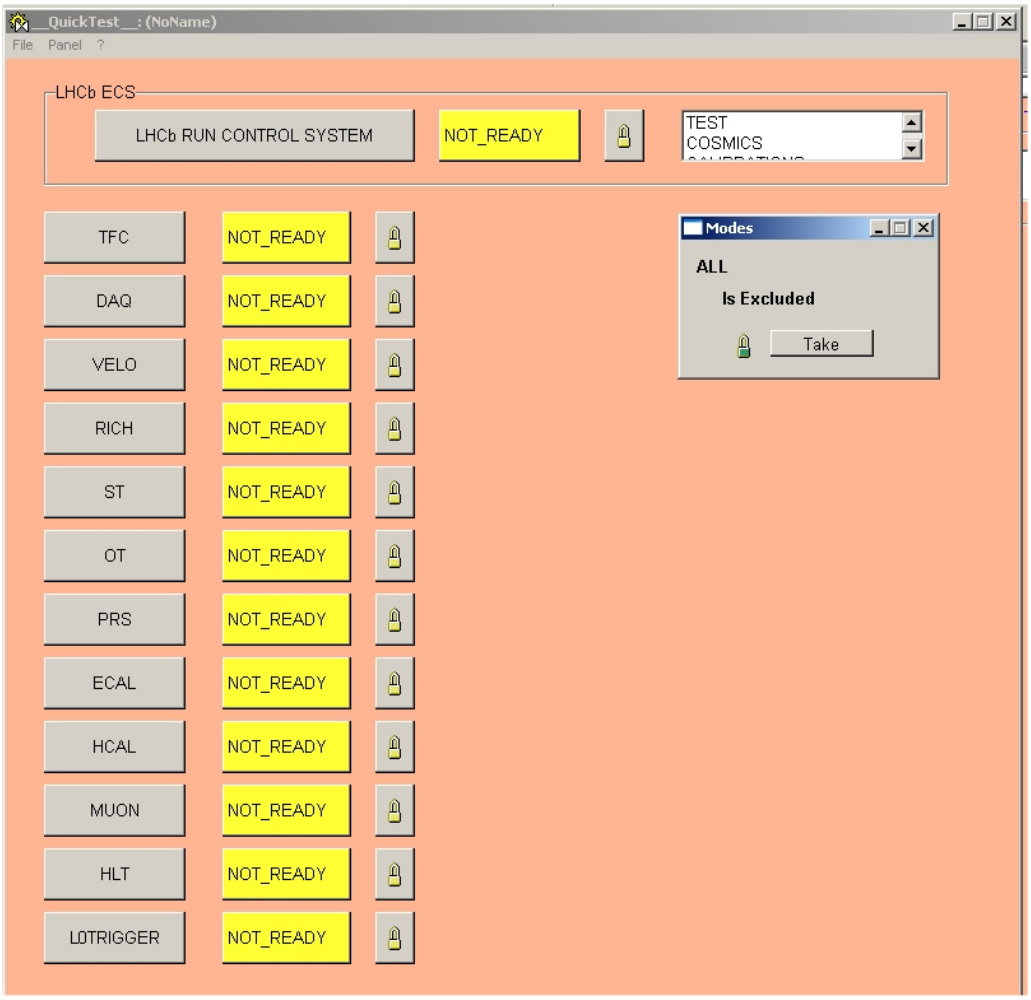


Figure 28. Very simplified LHCb RUN CONTROL PVSS panel.

1. The shift operator has to define a partition. Referring to Figure 28, it is done by clicking on the lock of the subsystems and click on “Take”. In Figure 28, the partition consists of the whole detector so “Take” has been clicked at the level of the RUN CONTROL SYSTEM (the top node).
2. Then the user selects a running mode and clicks on “Configure” as shown in Figure 29. As the state is NOT_READY, the only action is Configure referring to Figure 27. The selected configuration is “PHYSICS”.

Figure 29. Very Simplified view of the "Configure" command.

The action (or command) “Configure” covers all the operations to configure the equipment using the CIC DB.
This command is forwarded along the tree from the top node to the leaves. All the subsystems will load their settings related to “PHYSICS” from the CIC DB in PVSS and apply to the hardware using drivers. PVSS will send command to the hardware to check whether they are properly configured.

3. The children properly configured go to READY. If all the children are READY, the parent goes to READY. The state goes up to the top node by propagation of the state information along the tree. Figure 30 shows the case where everything has been configured properly into PHYSICS mode.

Figure 30. Very Simplified view of "Start the run".

1.4 Conclusion

In this chapter, we have presented the overview of the LHCb experiment. The detector is composed of different subdetectors with different tasks. They are composed of different types of modules. The Online system with its main components (TRIGGER, TFC, DAQ and ECS) is responsible for respectively selecting the most interesting events, synchronizing the equipment, forwarding the events' data from the detector to the permanent storage and to operate the detector.

The ECS architecture is based on PVSS and uses FSM toolkit to model the behavior of the experiment via states and actions. One of the essential steps of operating the detector is the configuration. The next chapter describes the configuration issues of the LHCb experiment.

References

[1] J. H. Christenson, J. W. Cronin, V. L. Fitch, and R. Turlay, Evidence for the 2π Decay of the K_2^0 Meson, 10 July 1964. In Physical Review Letters, Volume 13, number 4. p 138-140.

[2] LHCb Collaboration, LHCb Vertex Locator Technical Design Report. CERN/LHCC 2001-0011, LHCb TDR 5, May 31th 2001.

[3] LHCb Collaboration, LHCb RICH Technical Design Report. CERN-LHCC-2000-037, LHCb TDR 3, September 7th, 2000.

[4] LHCb Collaboration, LHCb Outer Tracker Technical Design Report. CERN-LHCC-2001-024, LHCb TDR 6, September, 2001

[5] LHCb Collaboration, LHCb Inner Tracker Technical Design Report. CERN-LHCC-2002-029, LHCb TDR 8, November 8th, 2002.

[6] LHCb Collaboration, LHCb Calorimeters Technical Design Report. CERN-LHCC-2000-036, LHCb TDR 2, September, 2000.

[7] LHCb Collaboration, LHCb Muon System Technical Design Report. CERN-LHCC-2001-010, LHCb TDR 4, May 2001.

[8] LHCb Collaboration, LHCb Trigger Technical Design Report. CERN-LHCC-2003-031, LHCb TDR 10, September 9th 2003.

[9] LHCb Collaboration, LHCb Computing Technical Design Report. CERN-LHCC-2005-019, LHCb TDR 10, June 2005.

[10] Jost B., Neufeld N., Raw-data format transport, LHCb Technical Note, September, 2004. EDMS 499933.

[11] CASTOR CERN Advanced Storage Manager, http://castor.web.cern.ch/castor/.

[12] J.-P. Dufey and R. Jacobsson, MEP Destination Assignment in the 1MHz Readout Scheme,

http://lhcb-online.web.cern.ch/lhcb-online/TFC/default.html#documents.

[13] Grid deployment at CERN, http://lcg.web.cern.ch/LCG/.

[14] V. Garonne, Étude, définition et modélisation d'un Système Distribué à Grande Échelle : DIRAC - Distributed Infrastructure with Remote Agent Control. PhD : Comp. University Aix-Marseille II : 2005, 192 p.

[15] C.Gaspar, M.Donszelmann, DIM a distributed information management system for the DELPHI experiment at CERN. In the Proceedings of the 8th Conference on Real-Time Computer applications in Nuclear, Particle and Plasma Physics, Vancouver, Canada, June 1993.

[16] C. Gaspar and B. Franek, Tools for the Automation of Large Distributed Control Systems, In IEEE Transactions on NUCLEAR SCIENCE, June 2006, Volume 53, Part III of three parts. Number 3. 14th Conference on real time (RT 2005) Stockholm, Sweden, June 4-10, 2005. p 974-979.

[17] M. Clemencic, J. Palacios, N. Gilardi, LHCb Conditions Database, in the Proceedings of the CHEP06 Conference, February 13-17, 2006.

[18] R.Barillere et al, Results of the OPC Evaluation done within the JCOP for the control of the LHC Experiments, October 1999. In the Proceedings of International Conference on Accelerator and Large Experimental Physics Control Systems, 1999, Trieste, Italy.

[19] CAEN, http://www.caeneng.com/news.htm.

[20] WIENER, http://www.wiener-us.com/.

[21] D.Breton, D.Charlet, P.Robbe, I. Videau, SPECS : A SERIAL PROTOCOL FOR EXPERIMENT CONTROL SYSTEM IN LHCB. In the proceedings of 10th ICALEPCS Int. Conf. on Accelerator & Large Expt. Physics Control Systems, Geneva 10-14 Oct 2005, WE1.5-4O.

[22] F. Fontanelli, G.Mini, M.Sannino, Zbigniev Guzik, R. Jacobsson, B.Jost, N. Neufeld, Embedded Controllers for Local Board-Control, IEEE Transactions on Nuclear Science, June 2006, Vol. 53, number 3, Part III of three parts. 14th Conference on real time (RT 2005) Stockholm, Sweden, June 4-10, 2005. p936-940.

[23] SMI State Management Interface, http://smi.web.cern.ch/smi/smi_2.html.

[24] Pierre-Yves Duval, Guide for ECS FSM design in LHCb detectors, LHCb Technical Note, May 9th 2006. EDMS 655828.

Chapter 2 Configuring the LHCb experiment

This chapter tries to give an outline of the task “configuring the LHCb detector”, a huge network of various devices. The first part describes the needs of the subsystems in terms of configuration. They are different as they use different module types. Combining different types of information such as connectivity and configuration parameters is also required to build a single-click control system. Secondly we explain what information is needed to configure the DAQ network. It includes routing tables, the configuration of the DHCP and DNS servers. Thirdly we present the concept of partition and its impact on the TFC system. Then we show the needs to manage the equipment so that the experiment can have a long lifetime. Then we explain how to handle fault detection and the strategy applied within LHCb. The ECS needs to make sure that the different subsystems are connected and properly configured. It should also make a decision if any problems occur while configuring. The introduction of autonomic tools is very convenient as it reduces human intervention. An autonomic tool is a self-intelligent agent or application which makes the updates automatically. The current commercial leader is IBM with BluePrint [1]. In this chapter, we also try to explain to what extent autonomic tools are used.
Finally we present briefly how performance measurement can be achieved.

2.1 Configuring the electronics

As seen in Chapter 1, a HEP experiment consists of between hundreds of thousands and millions of electronics modules of different types. All of them need to be properly configured.

2.1.1 New and different types of electronics

Experiments at LHC have integrated new types of devices and technologies in the design of the experiment. For instance, to interface the electronics to the control system SPECS (Serial Protocol for the Experiment Control System) and credit-card PCs are used. SPECS are essentially used for modules in radioactive area. SPECS is a protocol based on a 10Mbit/s serial link defined to be suited for the general configuration of remote electronics elements. It is a single master multi-slave bus. Credit-card PCs are embedded PCs used to provide the necessary local intelligence on an electronics board. They are connected to the central ECS via a conventional Ethernet and allow accessing the various components of the board.

Thus types of parameters such as SPECS addresses, FPGA codes, and registers of different sizes need to be set.

The type and the design of the detector technology and the electronics depend on the sub-detector.

For instance, the RICH detector uses HPDs (Hybrid-Photon Detector) [2] as shown in Figure 31. These devices need to be powered according to certain voltage and current settings.

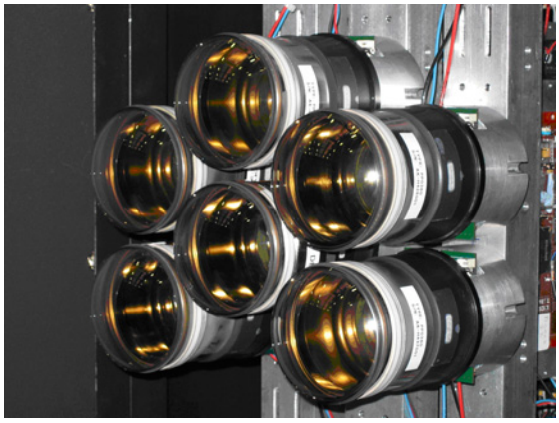


Figure 31. Six HPD devices in the RICH subdetector.

The VELO uses R- and Φ-sensors mounted on a hybrid [3] which have 16 beetles (chips) to configure as shown in Figure 32.

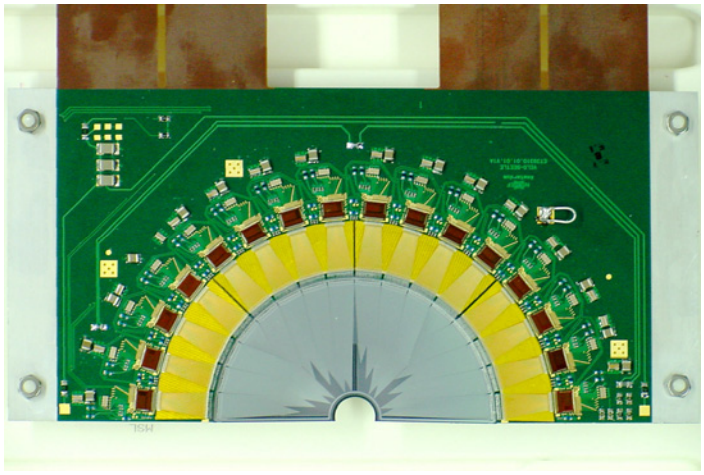


Figure 32. A VELO R-sensor with the 16 beetles chips.

2.1.2 A very large number of items to configure

The number of parameters to configure (and consequently the amount of data) depends on the type of devices.

For example, for the RICH1 and RICH2, the amount of data to configure for the L0 electronics is given by the Table 1:

Electronics type	**Number in RICH1**	**Number in RICH2**	**Amount of config data by electronics type (bytes)**	**Total RICH1 (Kbytes)**	**Total RICH2 (Kbytes)**
HPD	196	252	5125	1004.50	1291.50
L0 board	98	126	37.50	3.67	4.72
				1008.17	**1296.22**

Table 1. Amount of data to configure for the RICH system.

The IT and TT trackers for instance have less configuration data for the L0 electronics module as shown in Table 2:

Electronics type	**Number in IT**	**Number in TT**	**Amount of config data by electronics type (bytes)**	**Total IT (Kbytes)**	**Total TT (Kbytes)**
Beetle	1120	1108	20	22.400	22.160
GOL	1120	1108	1	1.120	1.100
Control cards	24	24	11	0.264	0.264
				23.784	**23.524**

Table 2. Amount of data to configure for the IT and TT systems.

The type of parameters depends on the device type as it is shown in the Table 3.

Board name	**Number of boards**	**component name**	**Parameters to configure**	**Number of components / board**
Hybrid	84	Delay chip	- 6*8-bit registers	1
Hybrid	84	Beetle	- 20*8-bit registers - 1*16 byte register	16
Control Board	14	TTCrx	- 3*8-bit registers	1
Control Board	14	SPECS Slave	- 3*8-bit registers - 4*32-bit registers	1
Temperature Board	5	Chip	- 1*64 bit register - 1*8-bit register	1
Repeater board	84	LV regulator	- 1*8-bit register	1
Tell 1 board	88	Channel	- Pedestal 1*10-bit - Threshold 2*8-bit - FIR 3*10-bit - Gain 1*14-bit	2048
Tell 1 board	88	FPGA code	- firmware	4
High Voltage power supplies	84	Commercial	predefined	
Low Voltage power supplies	84	Commercial	predefined	

Table 3. Amount of data to configure for the VELO system.

So the number of items and the quantity of items which need to be configured depends on the subsystem. It will have an impact on the execution time to load a

configuration for the whole experiment. As a reminder, the whole experiment should be configured in less than a few minutes.

The design of the subsystem in PVSS in terms of datapoint type structures will be affected. Shall all the details (registers for instance) be declared as datapoint elements? It is one of key point in modeling the control system of a subsystem in PVSS. The only way to solve the problem is to make some tests to compare the different representations

2.1.3 Using the connectivity to configure devices

In some subsystems, configuring the modules depends on the connectivity. For instance, in the HCAL subsystem, they need to configure PMTs [4], INT (Integrators) [4], LED [4], DAC [4] and FE [4] boards. A PMT transforms the light from the photons into electronic signals (photoelectrons). The LED emits light to the channel. It will allow calibrating the calorimeters and simulates the detector response. It is also used to control the linearity of the readout chains. The other three boards (DAC, INT and FE) are used to process the signals.

Figure 33 shows a simplified view of the HCAL connectivity. Each channel is connected to a PhotoMultiplier Tube (PMT) and two LEDs. A PMT is connected to a FE, an Integrator and a DAC board. The DAC boards power the PMT via HV. The INT boards measure the current from the PMT. It is for calibration purposes.

A LED is connected to a FE board and to a DAC (Digital Analog Converter) board. A DAC board can be connected at most to 200 PMTs and at most to 16 LEDs. FE and DAC boards process the electronics signal. A FE board can be at most connected to 32 PMTs.

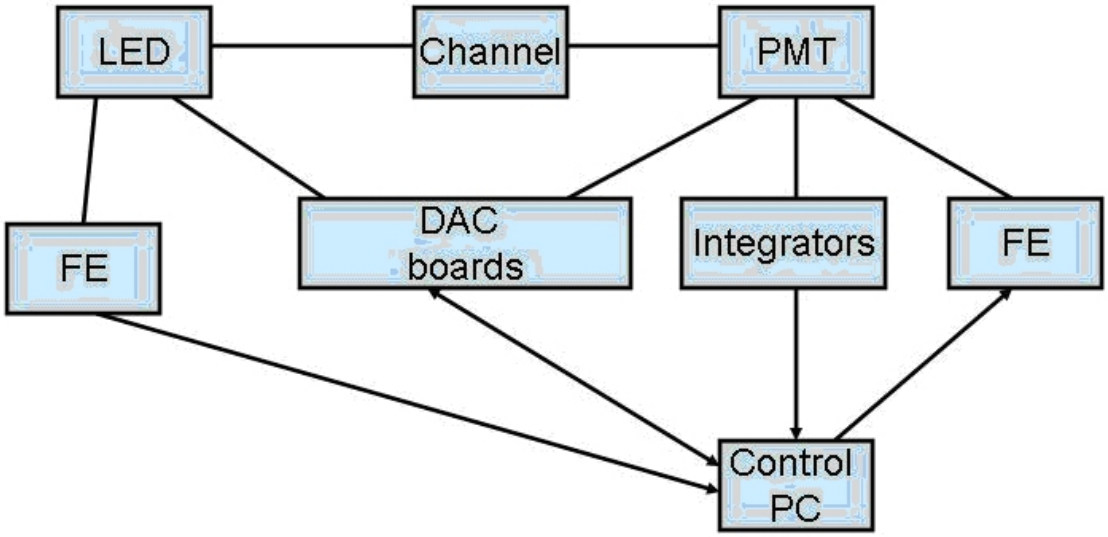


Figure 33. Simplified view of the HCAL connectivity.

To configuring the devices, the following information is required:

1. Info 1: The configuration and monitoring of the high voltage and the current of the DAC, INT and FE modules will be done via SPECS. They need to know the different SPECS addresses to communicate with the SPEC master which is located on a controls PC. So given a **channel name**, the respective

SPECS address for the DAC, INT and FE associated with it should be returned.

2. Info 2: The gain must be monitored. It is computed as follows $G=G_0HV^{\alpha}$. Typically G_0 and α will be properties of each PMT. A measurement will allow getting the value of the gain. If it is dropping, the HV needs to be adjusted. The calculation of the new HV needs to know what PMT it is connected to a given **channel name** as they know which channels are associated to a given DAC board. Then during a run the gain can be recomputed according to $HV'=HV\ (G'/G)^{1/\alpha}$.
3. Info 3 : Each channel will be illuminated by two LEDs. For calibrations purposes, they will ask which LED(s) illuminate the given **channel name**. Besides each link between a channel and a LED is associated with a quantity of light which is used for computations.

So configuring a module can also depend on its connectivity. It requires then a coherent and structured way to access the different types of information stored in the CIC DB.

2.2 Configuring network equipment

Another new type of equipment which is used in HEP experiments is network devices such as switches, routers, DHCP and DNS servers. Their configuration does not depend on the running mode.

2.2.1 The DAQ network (reminder)

The DAQ network has been described in Chapter 1. It is a Gigabit network based on IP. It consists of switches/routers and diskless farm nodes (PCs). There are two separate networks:

- The data network is used to route data traffic from the detector in the form of MEP packets from the TELL1 boards to the farm node and to send the most interesting events to permanent storage.
- The controls network is used to send controls commands such as start and stop devices, configure electronics, switches, routers and farm nodes (IP addresses, booting images for the farm nodes and the TELL1 boards, HLT algorithm for the farm nodes).

2.2.2 Network definitions

To understand better the needs of the DAQ in terms of configuration, some network concepts and definitions are introduced in the following sections.

2.2.2.1 IP packet and Ethernet frame

The Ethernet protocol [5] acts at the level 2 (Data Link) of the OSI (Open System Interconnected) Model [6], the IP protocol [7] at level 3 (Network).

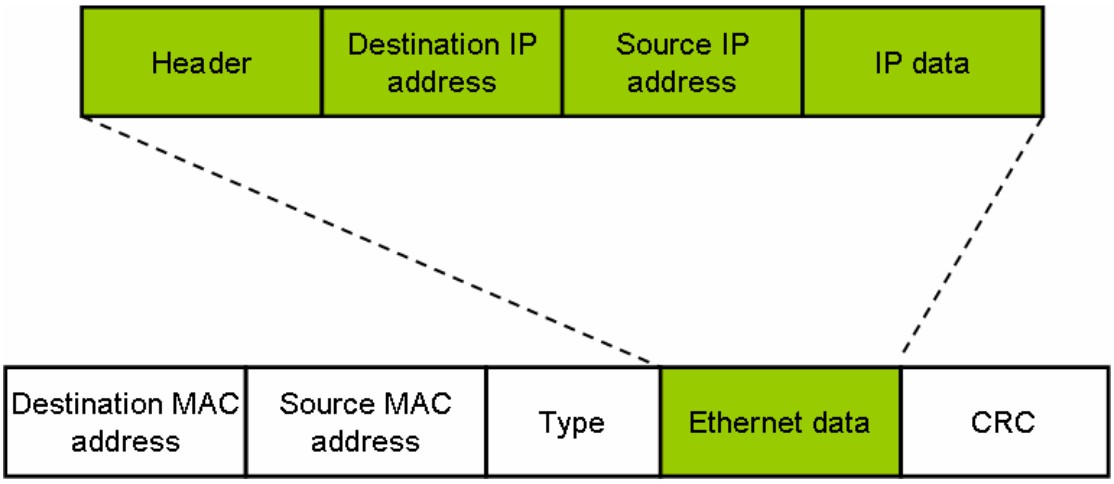


Figure 34. An IP packet encapsulated in an Ethernet frame.

An IP packet (see Figure 34) encapsulated in an Ethernet frame contains 4 different addresses, 2 for the sources (IP and MAC) and 2 for the destinations (IP and MAC). The destination addresses will allow identification whereas source addresses will allow reply. This means a communication can be established between the source and the destination. An IP address is coded with 4 bytes whereas a MAC address is coded with 6 bytes.

MAC addresses are uniquely hard coded and they are uniquely associated with a Network Interface Card (NIC). IP addresses are attributed using software. The size of Ethernet data is limited to 1500 bytes. Thus an IP packet may have to be split and sent in several Ethernet frames.

Broadcast addresses for Ethernet (resp. IP) are FF:FF:FF:FF:FF:FF (resp. 255.255.255.255)

In a network, equipment is identified both by IP and MAC addresses.

2.2.2.2 Hosts

Hosts are network equipment which can process data. TELL1 boards, PCs, which are respectively the sources and the destinations in the DAQ, are hosts as they build IP messages. Switches, routers are not hosts as they transfer the data. They do not build IP messages to send information.

2.2.2.3 Address Resolution Protocol (ARP)

ARP [8] is used to retrieve the MAC address of a given IP address. Referring to Figure 35, station A wants to send an IP message to station B. A knows the IP address of B but not its MAC address. It will broadcast an ARP request[3] for the IP address 194.15.6.14 to all the stations. Only B will respond by sending its MAC address. So A can send message to B.

[3] An ARP request consists of an Ethernet frame, with FF:FF:FF:FF:FF:FF and type=ARP

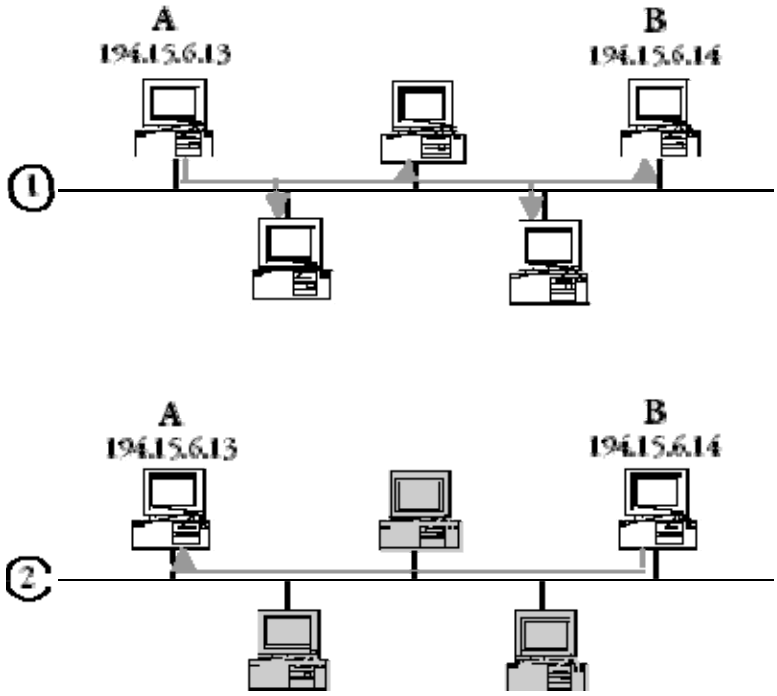


Figure 35.Illustration of the ARP protocol. The schema 1 shows station A which sends an ARP request to all the stations to get the MAC address corresponding to the IP address "194.15.6.14". The schema 2 shows that the station B answers to the station B because the ARP request was for him. It has the IP address "194.15.6.14". Shading means that the element is not active.

2.2.2.4 Subnet and IP Subnet

A subnet is a part of a network which shares a common address prefix. Dividing a network into subnets is useful for both security and performance reasons.

An IP subnet is an ensemble of devices that have the same IP address prefix. For example, all devices with an IP address that starts with 160.187.156 are part of the same IP subnet. This prefix is called the subnet mask.

2.2.2.5 Network Gateway device

A network gateway allows communication between two subnets (IP, Ethernet, etc.). A network gateway can be implemented completely in software, completely in hardware, or as a combination of the two. Depending on their implementation, network gateways can operate at any level of the OSI model from application protocols (layer 7) to Physical (layer 1).
In the case of an IP network, the gateway is usually a router. Its IP address is known by all the stations (PCs) of a same subnet.

2.2.2.6 IP routing (over Ethernet)

Routing is used when a station wants to send an IP message to a station which is not on the same subnet.

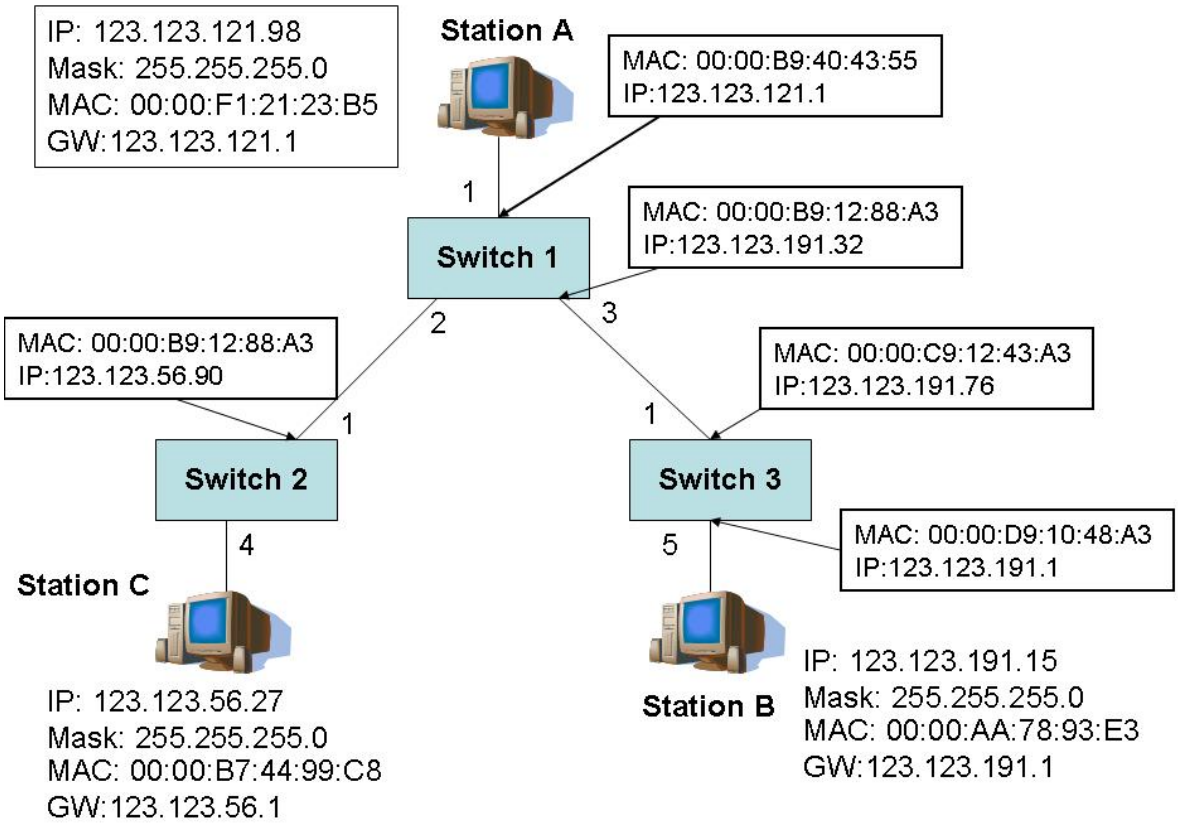


Figure 36. An example of IP routing.

Station A wants to send an IP message to station B. First A looks at the IP address of B. Referring to Figure 36, A is part of subnet 123.123.121 and B is part of subnet 123.123.191. Stations A and B are not in the same subnet. So A will send an IP packet to the gateway (Switch 1). A needs the MAC address of the gateway to build the Ethernet frame. A will look for the MAC address associated with 123.123.121.1 (IP address of the gateway) in its ARP cache. If it is not found, A does an ARP request for the MAC address of the gateway.

Then A sends the IP message to switch1. Switch1 examines the packet, and look for the destination address (123.123.191.15) in its routing table (see next definition). If it finds an exact match, it forwards the packet to an address associated with that entry in the table. If the router does not find a match, it runs through the table again, this time looking for a match on just the subnet part (in the example 123.123.191) of the address. Again, if a match is found, the packet is sent to the address associated with that entry. If not, it uses the default route if it exists. Otherwise it sends a "host unreachable" to the source.

In the example, Switch 1 will forward the message to Switch 3 via its Port 3. However, it needs to know the MAC address associated with the IP address of the next hop equals to 123.123.191.76 (found using its routing table) to build the Ethernet frame. It will look for it in its ARP cache. If there is no matching entry, it sends an ARP request.

Then Switch 1 forwards the message to Switch 3. It examines the destination address in the same way as Switch 1. Finally, the message arrives to B.

It is important to notice that the IP destination address of the message does not change during routing, unlike the destination MAC address. It is changed by the routers because it is the MAC address of the next hop.

2.2.2.7 IP routing table

An IP routing table is a table located in a router or any equipment which does routing. It is composed of several entries such as (we quote the most important ones):

- IP address of a destination (if it is equal to 0.0.0.0, it is the default route)
- Port number (of the router to forward the packet to)
- IP address of the next hop (if it corresponds to the destination address, it is equal to 0.0.0.0)
- Subnet mask of the next hop.

Figure 37 shows an extract of the IP routing table of switch 1.

Port Nb	IP address of the next hop	IP address of the destination	Subnet mask
1	0.0.0.0	123.123.121.98	255.255.255.0
2	123.123.191.76	123.123.191.15	255.255.255.0
3	123.123.56.90	123.123.56.27	255.255.255.0

Figure 37. An excerpt of the IP routing table of switch 1 (only the most important entries).

An IP routing must be consistent, i.e., the route to a destination must be uniquely defined if it exists in the routing table. So a destination address must appear only once in the routing table.

An IP routing table can be static, i.e. programmed and maintained by a user (network administrator usually).

Dynamic routing is more complicated and implies many broadcast packets. A router builds up its table using routing protocols such as RIP (Routing Information Protocol) [9], Open Shortest Path First (OSPF) [10]. Routes are updated periodically in response to traffic conditions and availability of a route.

2.2.2.8 Dynamic Host Configuration Protocol (DHCP)

This protocol allows a host which connects to the network to dynamically obtain its network configuration.

The DHCP server [11] will attribute an IP address, an IP name, a boot image location (set of files which will allow the host to get its configuration) to the newly connected host.

When a host starts up, it has no network configuration. It will send a DHCPDISCOVER message (special broadcast with IP destination equals to 255.255.255.255) to know where the DHCP servers are located. The DHCP server will respond by a DHCPOFFER (also a broadcast message as the host may not have an IP address) which suggests an IP address to the host (DHCP client). The host sends a DHCPREQUEST to accept the IP address. The DHCP server sends a DHCPACK to acknowledge the attribution.

The DHCP server can dynamically attribute an IP address or statically or both. It is fixed by the network administrator. If the address is attributed dynamically, it will be valid for a certain period. Moreover in the case of a dynamic attribution, it can take time or even fail (if all IP addresses are taken).

In case of a static attribution, the DHCP server has a dhcp config file defined by the network administrator which looks like Figure 38.

```
#
# Sample dhcpd.conf file
# Check with your network
# admin for local settings
#
# main Class B network
subnet 127.238.0.0
# subnet mask (Class B)
netmask 255.255.0.0 { not authoritative; }

group {
# gateway address
option routers 127.238.1.1;
# the boot image
filename "vmlinuz-2.4.26-ccpc08.nbi";
# the tftp-server IP address
server-name "127.238.142.63";address
# the nfs-server IP address
next-server "127.238.142.63";
option root-path
"127.238.142.63:/usr/local/defaultroot";
# here now the entries for the Hosts
host pclbcc02 {
hardware ethernet "01:02:03:04:05:05";
fixed-address 127.238.142.77;
   }
  }
```

Figure 38. Example of DHCP config file.

When a host sends a DHCPDISCOVER message, the DHCP server will look for the entry associated with the MAC address of the host in the dhcp config file which will contain all the information namely (referring to Figure 38):

- IP address which corresponds to the **fixed-address** information (which is static, always valid);
- IP name of the host which corresponds to **host** pclbcc02;

- IP address of the gateway which corresponds to **option routers;**
- IP address of the tftp-server given by **server-name** (from where to load the boot image);
- IP address of the NFS [11] server (to be used as a local disk) which is given by **next-server;**
- The boot image name which is given by **filename.**

At the beginning of the dhcp config file, some generic options are fixed. Options by IP subnets are inserted afterwards. Then groups are defined. A group is a set of hosts which have the same filename and server-name.

2.2.2.9 Domain Name System

A PC connected to Internet has at least one IP address and is part of a domain (for instance a CERN PC is part of the domain "cern.ch"). Working with IP addresses is not always very convenient. Associated with an IP address, a PC has also a host (or IP) name and aliases (optional).

A DNS server [11] is responsible of one specific domain. It performs the two following tasks:

- Given a host name, retrieve the IP address;
- Given an IP address, retrieve the host name and aliases if any (it is called reverse resolution). A DNS can distinguish between a host name and aliases as the host name is declared as the main one.

The DNS system helps in finding to which server a given URL (which corresponds to an IP name or an alias) points. It is organized as a hierarchy of PCs. For instance, a user wants to view the content of the URL www.wanadoo.fr. The PC sends a DNS query to search for www.wanadoo.fr. This query goes to the ISP (Internet Service Provider) DNS server (at home) or in the context of LHCb or other companies, the DNS server. If it knows the IP address (because it is already in the cache), it sends back the IP address. Otherwise it forwards it to the root DNS server. The root DNS server finds the URL is part of ".fr", it returns the IP addresses of the DNS servers (they are called the top level) responsible for the ".fr" domain, to the DNS server (the first one). Then it sends a request to one of the given DNS server responsible for ".fr", which sends back the IP address of the "wanadoo.fr" domain. As this URL was new, the first DNS adds it into its cache so next time, it will be able to send back the IP address of www.wanadoo.fr" immediately. In this example, we have stopped here because the IP address has been found. However if there are more sub-domains, we reiterate the previous process. If the IP address could not be found, we get an error, such as "Page could not be found". This mechanism is illustrated by Figure 39.

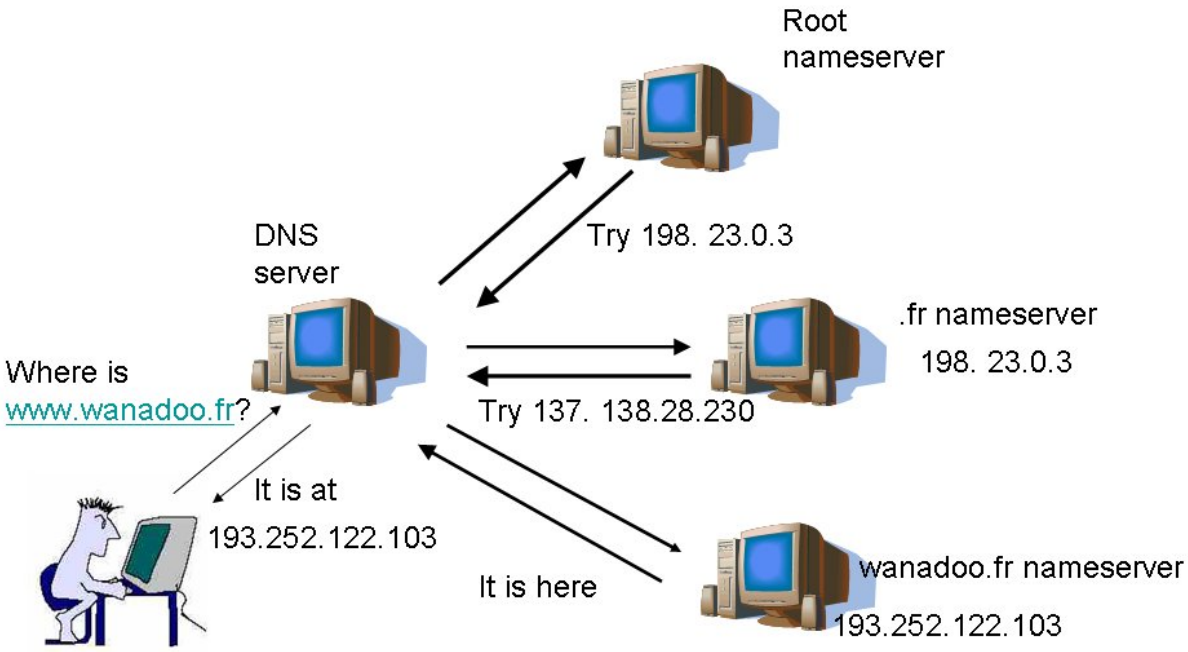


Figure 39. Principles of the DNS mechanism.

In the LHCb, there will be one disconnected domain (ecs.lhcb) and one authoritative DNS server with two other DNS servers, one which will be responsible for the DAQ equipment on the surface and another which will be responsible for the DAQ equipment in the cavern (underground).

Configuring a DNS server consists of providing two types of files.

- The forwarding file gives the IP address of a given host name. An example of this type of file is shown below:

```
$TTL    86400
# name of the domain "." is important              name of the
DNS server
ecs.lhcb.              IN            SOA          dns01.ecs.lhcb.
root.localhost. (
#some generic options
                                        200607130 ; serial
                                              3h ; refresh
                                            3600 ; retry
                                              4w ; expire
                                            3600 ; ttl
                                        )

# the given domain is supervised by this dns server if there
are several we # add the same line with the other names of dns
ecs.lhcb.   IN    NS    dns01.ecs.lhcb.

#name of the host without the "." and the corresponding IP
address
dns01       IN    A     10.128.1.1

sw-sx-01    IN    A     10.128.1.254
sw-ux-01    IN    A     10.130.1.254


# time01 is an alias to dns01 (main name)
time01            IN    CNAME dns01

slcmirror01 IN    A     10.128.1.100

ag01        IN    A     10.128.2.1
```

```
time02            IN    CNAME ag01

srv01       IN    A     10.128.1.2

pc01        IN    A     10.130.1.10
pc01-ipmi   IN    A     10.131.1.10
pc02        IN    A     10.130.1.11
pc02-ipmi   IN    A     10.131.1.11

dns01-ipmi  IN    A     10.129.1.1

slcmirror01-ipmi  IN      A       10.129.1.100


ag01-ipmi   IN      A       10.129.2.1
```

There following naming convention applies: the host name must be written without the domain name as it is appended automatically.

So if a machine has the following host name pclbtest45.ecs.lhcb, it has to be written as pclbtest45 and it should be followed by IN (internet) and then by A for Address if it is an IP address. For aliases, we give the alias name followed by IN then CNAME (canonical name) and finally the host name.

- The second type of file is the reverse resolver. It looks like as follows:

```
$TTL    86400
# IP address of the zone                    name of the dns
responsible
128.10.in-addr.arpa.      IN          SOA        dns01.ecs.lhcb.
root.localhost. (
                                          200607130 ; serial
                                                 3h ; refresh
                                               3600 ; retry
                                                 4w ; expire
                                               3600 ; ttl
                                          )


128.10.in-addr.arpa.     IN      NS      dns01.ecs.lhcb.

# part of the IP address                full host name
254.1                    IN      PTR     sw-sx-01.ecs.lhcb.

1.1                      IN      PTR     dns01.ecs.lhcb.
2.1                      IN      PTR     srv01.ecs.lhcb.


100.1                   IN      PTR     slcmirror01.ecs.lhcb.

1.2                      IN      PTR     ag01.ecs.lhcb.
```

In this type of file, the IP address is not written fully. In fact, an IP address is read from right to left and we delete the two last numbers. For instance, a PC has the following IP address, 123.45.67.89, the DNS reads it as 89.67.45.123. The IP address which specifies the zone is 123.45, so it becomes 45.123 when it is reverted. We take them off from 89.67.45.123, it remains 89.67 only. There should not be a dot at the end so that the IP address of the zone is automatically appended.

For the host name, the full name should be given and the dot should be added so that nothing will be appended to it.

In the LHCb network, it is foreseen to have one file per subnet and we have 4 subnets (two for the surface and two for the underground). They need to have an autonomic tool which automatically generates these files because it is tedious to write them manually (there are a lot of entries).

2.2.3 Network configuration

The network equipment in the DAQ network (routers, switches etc.) requires a specific configuration which is related to the connectivity.

Routing tables of switches will be configured statically for two reasons.

- Data paths should be deterministic, i.e. the routing path taken by a packet from a given TELL1 board to an EFF node should be known.
- It will avoid overloading the network with lots of broadcastings. As we have seen before, dynamic routing consists of many broadcast messages.

The ARP cache for the TELL1s, the EFF PCs and switches will be filled to reduce the number of broadcast messages.

Routing tables and ARP caches will be built using the information stored in the CIC DB.

The DAQ network structure will be similar to Figure 36. Station A will be a TELL1 board. There will be around 343 TELL1 boards connected to the core switch (Switch 1 in Figure 36). Switch 2 and Switch 3 will be distribution switches. Stations B and C will be Trigger Farm PCs. Each Sub-Farm will constitute an IP subnet.

In the DAQ system, IP attribution will be static to avoid any problems or time wasted at start up. The dhcp config file and DNS files will be generated using the information stored in the CIC DB.

Besides the network configuration, each port of a switch will have some configurable parameters such as speed, status, port type, etc. PCs will have some parameters such as the promiscuous mode, that is, Ethernet frames normally go to the above network layers only if they are addressed to that network interface. If a PC is put in the promiscuous mode, the Ethernet network interface (of the PC) will send all the frames (frames addressed to any host in the network), regardless of their destination address to the above network layers. It can be used to check that the network is properly configured.

All this information will be stored in the CIC DB.

For the DAQ, autonomic tools will be used to generate and update routing and destination tables. They will also be used to generate the DHCP config file and the DNS files. They are very convenient as there are a lot of PCs, switches and TELL1 boards which will get an IP address. Moreover an error in a routing table or in the DHCP config file or in the DNS system can mess up the network. Having automated tools which can fulfil this kind of task is very useful.

2.3 Configuring partitions for the TFC

Another concept which involves connectivity is partitioning from the TFC system point of view. A partition is the ensemble of modules of subsystems (or part of a subsystem) which will take data.

2.3.1 Impact on the TFC system

At the beginning of a new activity or run, the shift operator defines a partition.

In order to support a fully partitionable system, the TFC mastership has been centralized in one module: the Readout Supervisor. The architecture contains a pool of Readout Supervisors, one of which is used for global data acquisition. For separate local runs of sub-systems a programmable patch panel, the TFC Switch, allows associating sub-systems to different optional Readout Supervisors. They may thus be configured to sustain completely different timing, triggering, and control. The TFC Switch distributes in parallel the information from the Readout Supervisors to the Front-End electronics of the different sub-systems.

2.3.2 Programming the TFC switch

The TFC Switch incorporates a 16x16 switch fabric. Each output drives one subdetector such as RICH1, RICH2, VELO, etc, and each input is connected to a separate Readout Supervisor. In other words, it means that all the TELL1 boards which are part of a same subdetector will be driven by the same output of the TFC Switch. This switch is programmed according to the selected partition.

Let us consider the following example. The shift operator chooses VELO, RICH1 and RICH2 as a partition.

Programming the TFC Switch consists of two steps:

- Find the output ports which are connected to the subsystems in the partition (VELO, RICH1 and RICH2 in the example).
- Find the input port which is connected to the selected Readout Supervisor (usually the first free Readout Supervisor is chosen).

Figure 40 illustrates the concept. The Readout Supervisor 1 has been selected to control the partition {VELO, RICH1, RICH2}. Red components mean that they are used for the data taking.

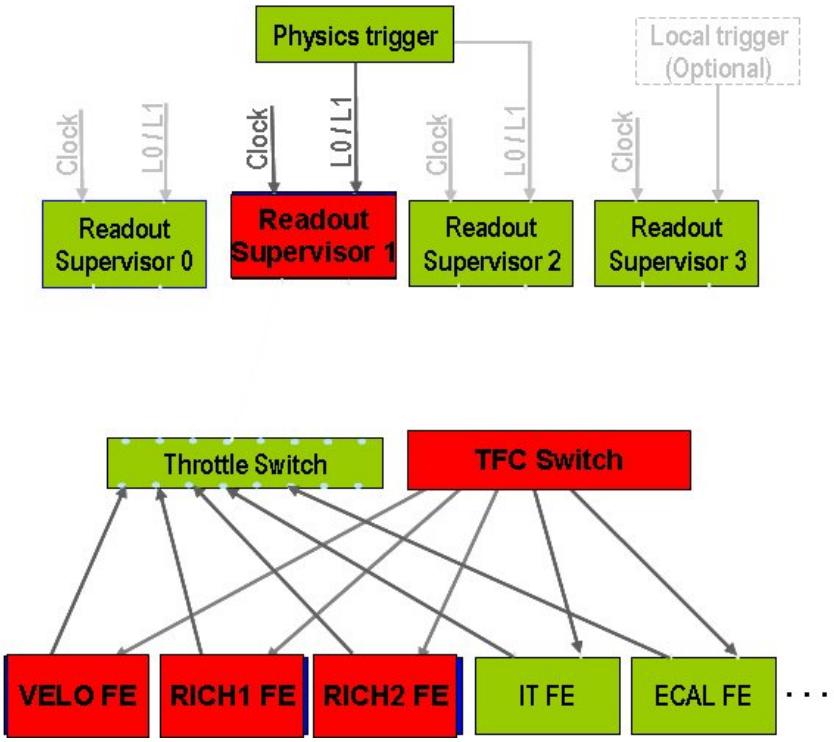


Figure 40. Handling the partition in the TFC system (first step).

Then using this information, the TFC switch is programmed as shown in Figure 41 (links in green).

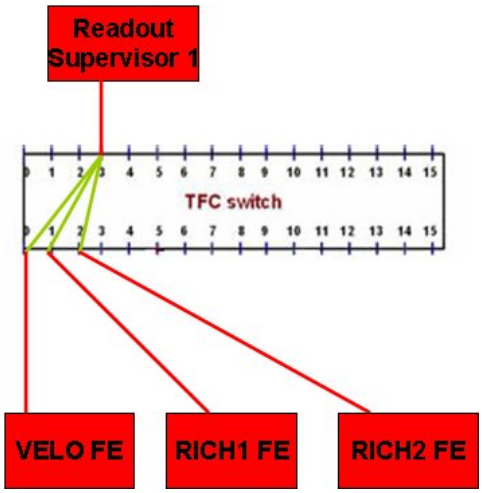


Figure 41. The TFC internal connectivity (second step).

Last of all the Readout Supervisor is configured according to the specific activity.

2.3.3 Subsystems from the FSM view

In Chapter 1, we have explained that from the controls point of view, the LHCb experiment will be modelled as a hierarchy and its behaviour and its states will be implemented using a FSM. Subsystems can be selected by clicking on them from a PVSS panel. Another panel will show up and displaying the decomposition of this subsystem. For instance, clicking on VELO will pop up another panel showing that the VELO is split into two parts, VELO_A and VELO_C. This principle is iterative,

i.e., by clicking on VELO_C, its different parts appear. It stops when displaying the electronics modules.

2.3.4 Subsystems from the TFC view

Using the FSM view, the shift operator can define a partition with half of the devices of VELO_A and another partition with other half of devices of VELO_A.

Although theoretically possible, this cannot work. The granularity of the parallel partitions is fixed by the TFC system, especially by the number of outputs of the TFC switch. In section 2.3.2 Programming the TFC switch, we have seen that the readout supervisor is responsible for one partition. And via an output port of the TFC switch, it sends the signal to a set of electronics module part of a certain ensemble of a subsystem. This "certain ensemble" is the limit of parallel partitioning. In other words, this "certain ensemble" cannot be split into several parts to form different partitions. For instance, referring to Table 4, two parallel partitions can be defined out of the VELO, one consisting of the electronics module of the VELO_A and another one consisting of the electronics module of VELO_C. But it is not possible for instance to have one partition with electronics modules of half of the RICH1 and another partition with electronics modules of the other half of the RICH1 as they are driven by the same TFC output port.

Subsystem name (as displayed to the user in the FSM top view)	**Subsystem name in the TFC (defines an upper limit on the number of simultaneous partitions)**
VELO	VELO_A and VELO_C
L0TRIGGER	PUS, L0CALO, L0MUON, L0DU
RICH	RICH1 and RICH2
ST	IT and TT
OT	OT
ECAL	ECAL
HCAL	HCAL
PR/SPD	PR/SPD
MUON	MUON_A and MUON_B

Table 4. Subsystem names and their decomposition.

2.4 Equipment management

The LHCb detector will be used to take data over years. Equipment will be swapped, replaced, etc.

To allow the detector to run in the best conditions, an inventory of the equipment and tracing back each replaceable device is essential. Also it should be possible to reproduce the configuration that a detector had at a given time.

The time reference for the device history is when a device arrives at LHCb.

2.4.1 Device status

Each device (included replaceable device components such as a chip) has a status and a location which can evolve with the time. For instance a device can be a spare, in use. Also it can be in repair or even destroyed. In some cases, it can be taken out for test purposes. The full list of statuses will be explained in detail in the next chapter.

2.4.2 Allowed transitions of status

Transitions between one status and another one must be clearly specified. It is quite intuitive that if a device is destroyed, it cannot be used any longer. So it cannot go to another status. Another case is when a device fails; it cannot be replaced with a device which is being repaired. That is why it is very important to define the transitions associated with the actions which must be performed to ensure data consistency. And the use of autonomic tools is very helpful in equipment management as it is easy to make mistakes.

2.4.3 Inventory

The inventory consists of:

- Sorting devices per status at a given time. It means at time T, one should be able to know where the device is and what status it has.
- Updating the status of the device and making the necessary changes associated to the status change. It is important to keep consistency in the database. For example, if a device breaks, it will be replaced by a spare. So the status of the broken device changes and goes to something like “being repaired”. And the spare which replaces it, is no longer a spare and goes to something like “is being used”. Also it is important to update the statuses of the components of a device in a consistent way. If a device breaks and needs to be repaired, its status is IN_REPAIR. Its components will also be IN_REPAIR.

2.5 Fault detection and verification of the correctness

The commissioning phase is an important step during the installation of the detector. During this phase, all the electronics modules are tested and certified to work properly when they are integrated with each other.

2.5.1 Verifying the configuration of the modules

It is important to check that devices are configured properly. To achieve this, the following policy has been applied at LHCb. The different steps have been presented in Figure 42. There is an automatic read-back mechanism of the values written in the hardware using DIM. If the device is properly configured, it goes to state READY, if not it goes to state ERROR. Then the FSM will try to recover the system. In the future, when the LHCb detector is fully operational, some automatic recovery actions will be taken based on the type of errors. The set of tools that come along with the CIC DB allows building an autonomic control system. For instance, the FSM can get the history of a faulty module to check if this kind of failure already occurs and consequently react properly.

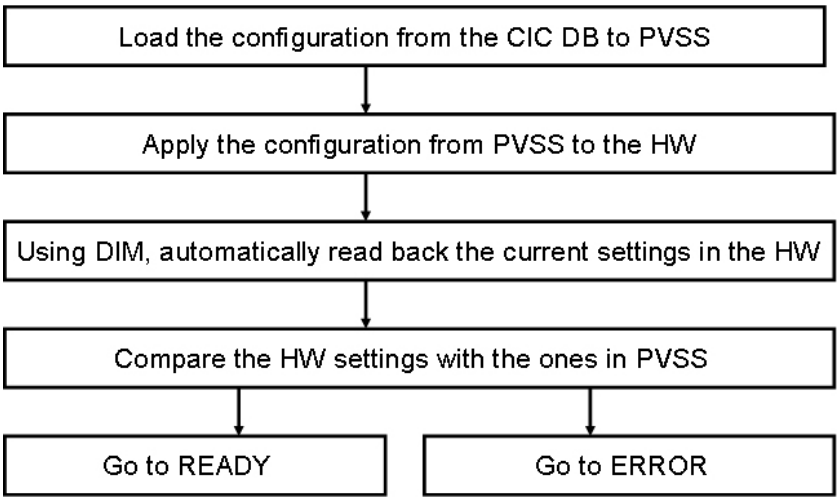


Figure 42. Checking that the device is properly configured.

2.5.2 Tests of links

2.5.2.1 Issues

LHCb is a big collaboration of several European institutes. Each member contributes in building and implementing part of the LHCb equipment. Integration and installation of all the pieces will begin at CERN. All the different electronics will be connected together. During this phase, connectivity needs to be tested. Typically it means that the electronics people want to know:

- Referring to Figure 33, a HCAL_PMT should send a data signal to a HCAL_DAC board. Get all the electronics devices between a given HCAL_PMT and a given HCAL_DAC to determine which one(s) can be faulty.
- A board A should receive data from a board of type VELO_TELL1. Get all the paths (in details) between board A and boards which are of type VELO_TELL1.
- Referring to Figure 43, GOL1 (Gigabit Optical Link, it's an optical driver) sends data to which FPGA(s).

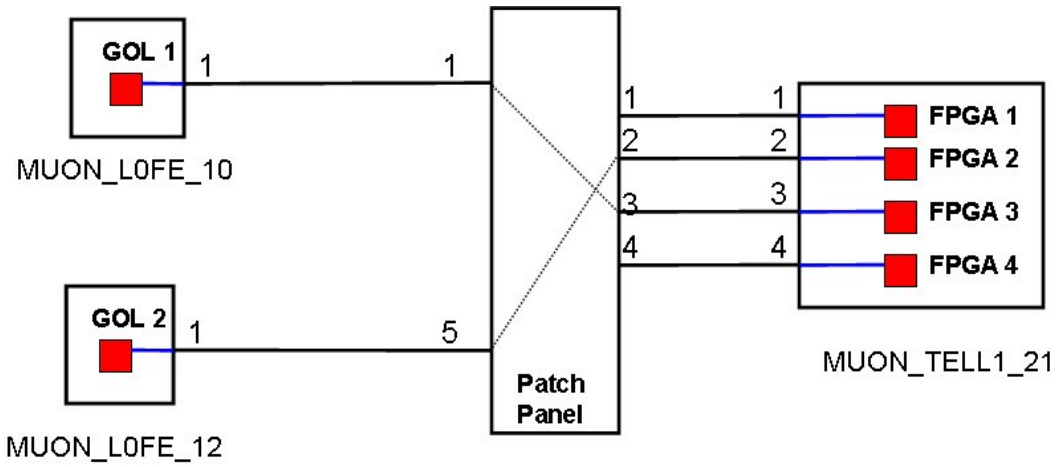


Figure 43. Example of internal connectivity.

2.5.2.2 Macroscopic and microscopic connectivity

From the previous examples, we can see that there are two levels of connectivity:

- Macroscopic connectivity which describes the physical links (wires) between devices.
- Microscopic connectivity which describes the connectivity of a board itself, i.e. between board components. For instance, referring to Figure 44, the repeater board has an internal structure that can be described. It is composed of 4 driver cards, a LV mezzanine and an ECS mezzanine. The driver card 1 is connected to j4 of the repeater board on its input and j20 on its output, etc.

In principle, each subsystem will save its own connectivity at the macroscopic level. In total there will be roughly one million macroscopic links.

Connectivity of a board, i.e. microscopic connectivity will be saved if necessary, depending on the use cases.

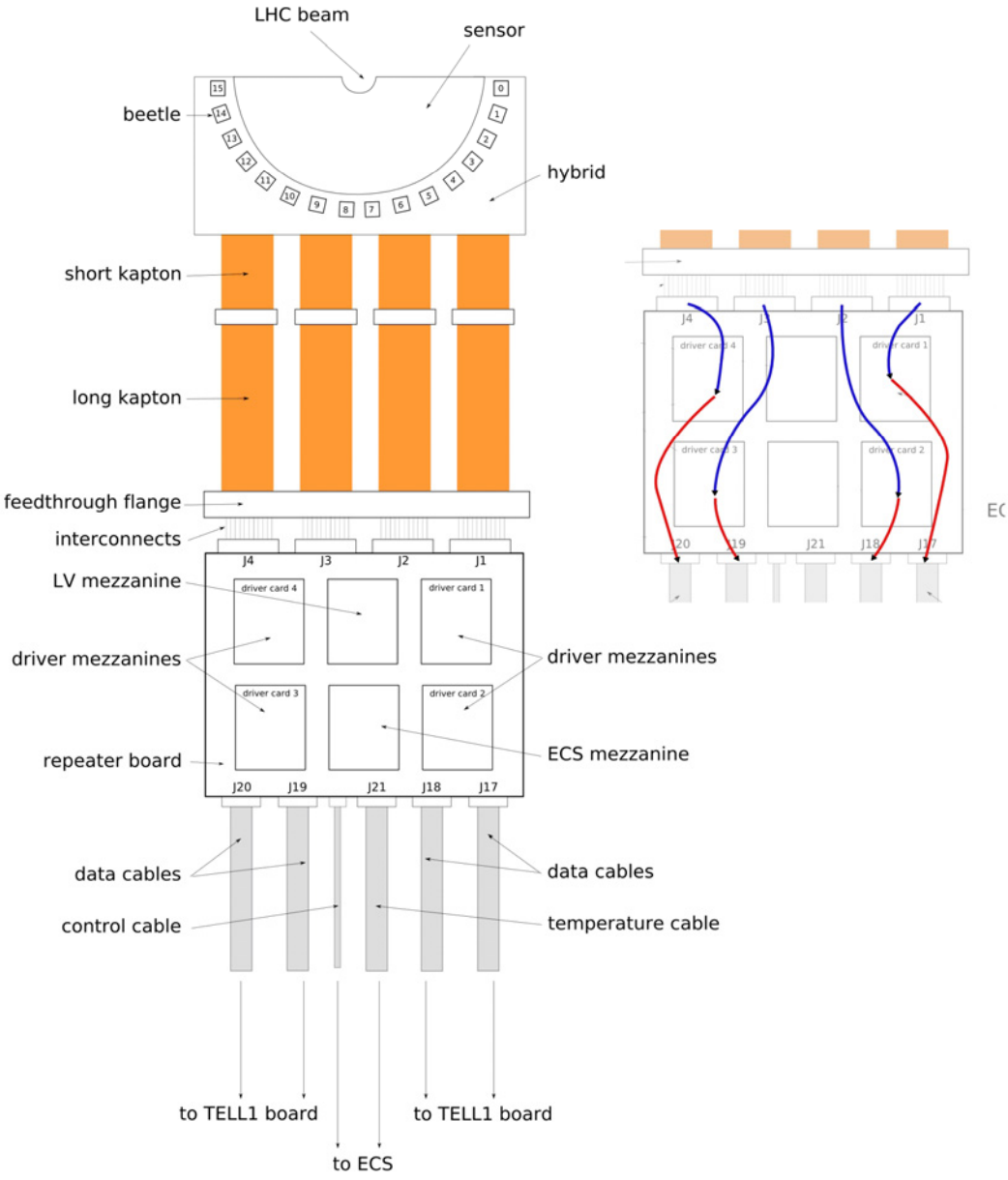


Figure 44. An slice of the VELO connectivity, from a hybrid module to the TELL1 board. On the right, there is the internal dataflow of the repeater board.

2.5.2.3 Internal connectivity of a board

The internal connectivity of a board consists of describing the output ports which can receive data from a given input port of a device due to an architecture constraint (it is fixed). In most cases, in LHCb, there is no need to store the internal connectivity of a device if the latter does not contain a microscopic component. For instance, the internal connectivity of the TFC switch or a DAQ router is set dynamically using destination or routing tables. In principles any input can send data to any output ports. However, there are some devices which have a special connectivity.

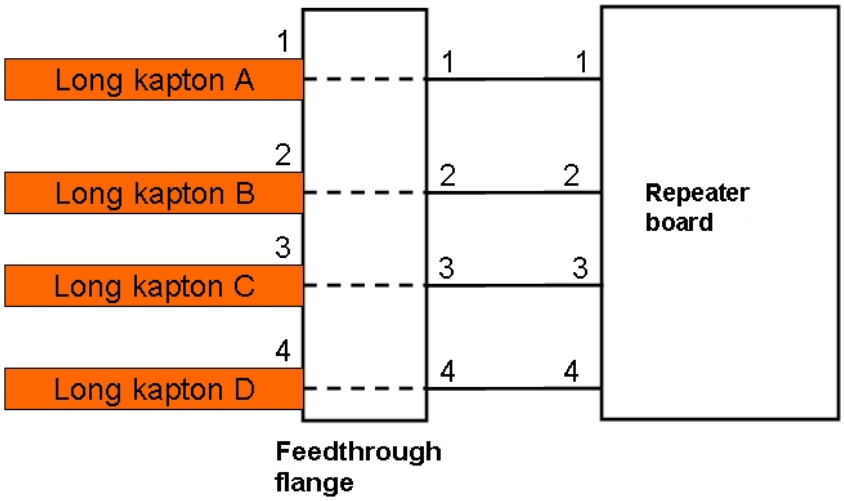


Figure 45. The internal connectivity of the feedthrough flange.

Figure 45 shows the internal connectivity of the VELO feedthrough flange. It is also shown in Figure 44. A signal coming at the input 1 of the feedthrough flange can only go out from the output 1.

So the combinations (input, output) of this device are not all valid. There is a need in that case to store the internal connectivity so that we do not get paths between the Long kapton A and the input port 4 of the repeater board.

2.6 Performance measurements

The following performance measurements were carried out using benchmarks (we focus on the configuration software):

- The maximum number of electronics that a controls PC can configure;
- The best architecture in terms of building the hierarchy of controls PCs;
- The best representation of a type of information in the CIC DB in terms of execution time (for requests);
- The fastest function implementation in the CIC_DB_lib;
- The upper limit of concurrent users to the CIC DB without affecting the performance.

2.7 Conclusion

In this chapter, we have described the different steps needed to configure a detector. It is quite a complex procedure as there are a lot of electronics modules of different types to be represented. Also connectivity and configuration parameters have to be related to configure devices such as for the Calorimeters.

Since the modules are built from different places, there is also a need to verify and test the integration of all the modules. The detector has a long lifetime and its equipment should be maintained. It requires an inventory and storing the history of devices.

All the information related to configuration, connectivity and history/inventory of devices will be modelled in the LHCb CIC DB, considered as a central repository of information about the detector.

Errors or user mistakes can be easily made. A policy of implementation has been applied to verify that a device is properly configured based on an automatic read-back hardware values mechanism. For example, a user can forget to update the connectivity of a device if the latter fails. Or if a link breaks in the DAQ network, one has to manually change the routing tables of switches. Beside as there are thousands of links and hundreds of switches, it implies a lot for work to update all this information. Performing all these operations manually is tedious and bound to errors. Thus the tools developed must be as much autonomic as possible. This is the guideline which has been adopted by the LHCb Computing group.

References

[1] IBM Research, An architectural blueprint for autonomic computing, White Paper. Available: http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf.

[2] A. Braem, E. Chesi, F. Filthaut, A. Go, C. Joram, J.Séguinot, P. Weilhammer1) and T. Ypsilantis, The Pad HPD as photodetector of the LHCb RICH detectors. LHCb Note, October 1999. LHCb 2000-063 RICH.

[3] LHCb Collaboration, LHCb Vertex Locator Technical Design Report. CERN/LHCC 2001-0011, LHCb TDR 5, May 31th 2001.

[4] LHCb Collaboration, LHCb Calorimeters Technical Design Report. CERN-LHCC-2000-036, LHCb TDR 2, September, 2000.

[5] Ethernet Protocol IEEE 802.3. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specification, 2002. http://standards.ieee.org/getieee802/download/802.3-2002.pdf.

[6] ISO/IS 10040, Information Technology - Open Systems Interconnection - Systems Management Overview, August 1991.

[7] Internet Protocol, DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION, RFC 791, September 1981. http://www.ietf.org/rfc/rfc0791.txt.

[8] An Ethernet Address Resolution Protocol, RFC 826, November 1982. http://www.ietf.org/rfc/rfc0826.txt.

[9] Routing Information Protocol, RFC 1058, June 1988. http://www.ietf.org/rfc/rfc1058.txt.

[10] OSPF Version 2, July 1991.http://www.ietf.org/rfc/rfc1247.txt.

[11] Douglas E. Comer., Internetworking with TCP/IP, Vol I: Principles, Protocols and Architecture Third Edition, Upper Sadler River, New Jersey: Ed. PRINTICE HALL, 1995. 613 p.

Chapter 3 Software architecture

In chapter 1, section 1.3.1, the ECS software architecture has been described. In this chapter, we will focus on the implementation of the CIC DB and its tools, which are the part of the ECS architecture responsible for configuring the experiment.

First, this chapter presents an overview of the software architecture that has been built and integrated in the ECS software architecture. It is a 3-Tier architecture, with a database, object and GUI layers. Secondly, we describe the three layers. Then we explain the importance of the API in building autonomics tools. Finally, we discuss the choice of such an architecture.

3.1 Outline of the architecture

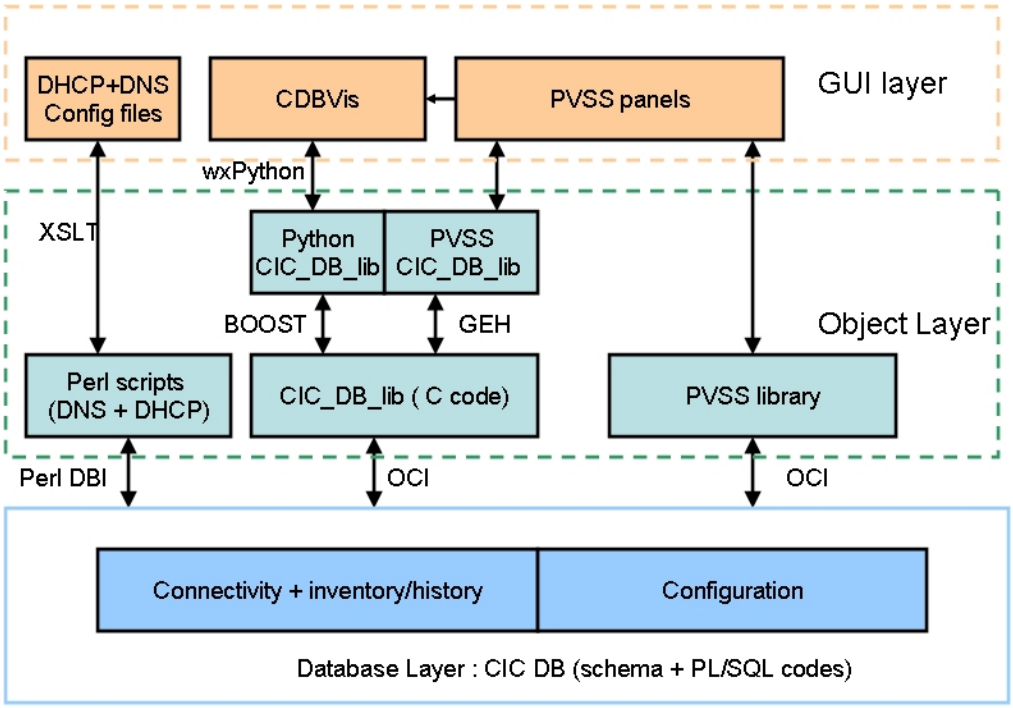


Figure 46. Software architecture.

Figure 46 represents the 3-Tier architecture which has been implemented to handle the configuration issues for the ECS. The 3-Tier architecture has the advantage to be robust and scalable. Changes on one of these layers do not affect the others unless a complete modification in the table design or in the API is made.
The next section explains the 3 layers in more detail.

The GUI layer, via PVSS panels, allows the integration within the ECS system.
The aim of this architecture is to provide the following features:

- Provide a database schema which is complete so that the ECS can find all the information required to configure the equipment properly namely configuration settings, connectivity of a device and history/inventory of the hardware;
- Provide a database schema which caters for the whole experiment to permit to reduce the number of tools to be implemented and to make the software maintenance easier;

- Provide an efficient schema and a fast PVSS CIC DB interface to allow loading the configuration of all the equipment in less than one minute;
- Provide a set of adaptive and autonomic tools to allow reliable and consistent manipulation (insertion, query, update and of the data by non-expert DB users;
- Provide a graphical tool to allow navigation through the content of the CIC DB to permit fault detection or inconsistencies in the database.

3.2 Description of the 3-Tier architecture

3.2.1 Database layer

All the information regarding configuration will be saved in one single relational database, the CIC DB. It is implemented using Oracle technology. The database consists of two parts:

- A common part for the 4 experiments at the LHC, which contain information about configuration parameters of devices and the hierarchy of devices. The schema and PL/SQL scripts for this part are provided by the CERN PVSS Support group. To make their design, they have obtained use cases and requirements from the 4 experiments [1].
- A part specific to the LHCb experiment which contains the connectivity of the subsystems and history/inventory information. The design has been done as part of my thesis. The methodology applied is described in the next chapter. It consists of identifying the users, collecting the requirements and use cases and building the schema using the ERM. This part contains PL/SQL codes too. There is one PL/SQL package (**routingtable_pck**) which generates and updates routing and destination tables. A set of some PL/SQL functions which were used when the SQL queries were too complex (especially regarding equipment management). As mentioned in the previous chapter, the DAQ is composed of thousands of items. The PL/SQL applications (such as **routingtable_pck**) help in building an autonomic control system by automated some steps further to a change in the content of the database.

To ensure consistency between the two parts (common part and LHCb specific part), the name of the modules should be the same. Guidelines and presentations have been given to make the LHCb collaboration aware of this constraint, which is very intuitive. Apart from the name of the modules, the two parts contain disjoint information so there is no need of a communication between the two parts.

The database schema of both parts is described in Chapter 5. The implementation of the PL/SQL package to create the routing and destination tables and other PL/SQL codes is described in Chapter 6.

3.2.2 Object layer

This layer is the core of the 3-Tier architecture as it links the two other layers. It has knowledge of the runtime values and is aware of the database structure.

The object layer is composed of three parts (referring to Figure 46):

- The PVSS scripts provided by the CERN PVSS Support group which allow manipulating data related to the configuration parameters (saving/loading a

configuration). It is a PVSS library which communicates with the database layer via OCI. This tool is described succinctly in Chapter 7.

- The CIC_DB_lib is a C-library which uses OCI as a DB interface and which provides autonomics features by reducing the human intervention. This library consists of a set of functions which permits to insert, select, delete and update information related to the connectivity and inventory/history in a consistent and secure manner. It is based on an API which corresponds to the needs (use cases and requirements) of the CIC DB users. Some of the functions embed PL/SQL functions. Two bindings have been implemented, one in Python using BOOST and one in PVSS using the GEH (Generic External Handler). This library has been implemented according to the autonomics principles. It should be robust against input errors and smart by reducing human intervention. The detailed implementation of the CIC_DB_lib and its two bindings is explained in Chapter 7.
- The Perl scripts which create the DHCP config file and the DNS set of files using the connectivity information. They communicate with the database layer via PERL DBI [2]. These two scripts also take part in building an autonomic control system by dynamically creating the files. The structure of these 2 scripts is described in Chapter 7.

The two last items were part of my work, as they are specific to LHCb.

3.2.3 GUI layer

This layer displays the data to the users. It is the top layer. It also consists of three parts:

- The PVSS panels which are designed by the subdetector groups. They use the functions provided by the PVSS CIC_DB_lib binding and the PVSS scripts provided by the CERN PVSS Support group. These panels allow configuring and monitoring one or a hierarchy of devices. Some examples of panels will be given in Chapter 8.
- The CDBVis graphical editor which allows navigating through the content of the CIC DB (connectivity essentially). It uses the Python CIC_DB_lib binding. This tool enables the users to insert and to view their connectivity. It uses wxPython [3] as python graphical module. The outline of CDBVis is given in Chapter 8.
- DHCP and DNS files which are the output of the Perl scripts. This set of configuration files allows verifying that the content of the database is not incoherent. They have been generated using XSLT [4].

3.3 The API

One essential step in implementing this architecture is defining an API which should be complete so that all the users of the CIC DB can save and retrieve the information needed. To achieve to build a complete API, a list of use cases has been written down and has been presented for validation. The API permits to the user to communicate with the CIC DB without any knowledge of the database schema and SQL or PL/SQL language.

The main problem was the timescale. The subdetectors did not have a clear view of their use at the same time as some of them experienced delays in the productions of

their electronics modules. So the content of the API has been extended over the months.

3.4 Choice of the languages

In this section, the different choices of the languages and technologies used to build the three layers are discussed.

3.4.1 Oracle and PL/SQL

The CIC DB is an Oracle database because there was a strategic decision by LHCb to use Oracle for all its database. This simplifies maintenance and the required expertise inside the experiment. As there is a HEP wide Oracle license, outside users can do their development without extra costs. The alternative would have been mySQL but the range of tools and features is limited compared to Oracle and the two products are not compatible.

For efficiency reasons, PL/SQL has been used to embed complex SQL queries to avoid transferring long statements through the network. It has been also used to generate and create the routing and destination tables because all the SQL executions are performed on the server-side. Thus it avoids loading information from the database, processing them to generate routing and destination tables in the client-side and putting back the results in the CIC DB.

3.4.2 Use of C and OCI, Python and Perl

At LHCb, C, C++ and Python are the most common languages used. There is no use of Java.

The CIC_DB_lib had to be implemented in one of these three languages. It had to be efficient, fast and possible to build a PVSS binding (to be integrated in the ECS).

As the Python DB interface is built on top of OCI, it is clumsy to implement CIC_DB_lib in Python. Moreover there is no direct way to integrate Python in PVSS, other than making a binding in C.

So there is C or C++ left for implementing CIC_DB_lib. Building a PVSS and a Python binding from a C-library is feasible, but building a PVSS binding from a C++-library would have been tricky as PVSS does not use classes and methods. Moreover the main inconvenience of building a C++ library is memory management (memory allocation for classes). Finally OCCI (best DB interface for C++) [4] is built on top of OCI which is the native DB interface of Oracle so from a performance point of view OCI is the best Oracle interface.

Thus for all these reasons, CIC_DB_lib has been implemented in C with two bindings one in Python, and another one in PVSS and it uses OCI as a DB interface. The main problem to cope with is versioning compatibility, which is explained in Chapter 7.

Another alternative was to use DIM instead of GEH. The input and return values would be transferred using DIM between PVSS and a CIC_DB DIM server. In fact, the very first implementation was using DIM to communicate with the CIC DB, but the implementation based on GEH was faster in our context. So it restricts the access to the CIC DB to PVSS only.

Perl has been used to generate the config files because it is a common script language. They can be triggered from some PVSS panels or called from a C code. Python could also have been used.

3.4.3 Use of BOOST, XSLT and GEH

The only way to interface a C-library to PVSS is the GEH. The GEH allows implementing controls libraries. It works both on Linux and on Windows.

To make the Python binding, BOOST has been used. Implementing a Python wrapper directly would have been painful and hard to maintain as the CIC_DB_lib contains a lot of functions. Another possible and simpler wrapper would have been SWIG [5]. But I experienced some problems to convert C pointers which were both in-out parameters in the functions. There is an example in chapter 7. Finally I decided to use BOOST as it was integrated in the LHCb Offline software environment via Gaudi [6]. BOOST on its own is a powerful application but quite complex. Using it via Gaudi avoids maintenance problems especially for new versions of BOOST which need to be recompiled with the Python library. It can then cause some incompatibilities with the Python version used at LHCb.

XSLT has been used to convert the XML files (produced by the Perl scripts) into DNS and DHCP config files. The details of this conversion are given in Chapter 8. It is flexible and offers a wide range of features (the output can be formatted as needed). As the DNS and DHCP files follow strict syntax containing specific user options, the XSLT permits to check the syntax of the XML files and to make sure that the user does not provide incompatible options.

3.5 Conclusion

This chapter describes the software architecture implemented to handle configuration. It is 3-Tier architecture. The top layer (via PVSS) ensures the integration within the ECS architecture. The third layer consists of the CIC DB as a database is a persistent technology to store data. The second layer which is the bridge between the two previous offers libraries to manipulate the data in the CIC DB according to the users' needs.

The architecture has been designed and implemented to help the LHCb ECS to be an autonomic system by reducing the human intervention and automated steps further to a change in the content of the CIC DB.

To make the design of the three layers properly, identifying the users of the CIC DB, collecting the requirements and use cases are essential and will be explained in the next chapter.

References

[1] JCOP Framework, Configuration Database Tool, User and Development Guide, June 2004.
http://itcobe.web.cern.ch/itcobe/Projects/Framework/Download/Components/Configuration/fwConfigurationDBUserDevelopmentGuide.pdf.

[2] Perl DBI, http://dbi.perl.org/.

[3] wxPython module, http://www.wxpython.org/.

[4] XSLT, http://www.w3.org/TR/xslt.

[5] ORACLE. OCCI (Oracle C++ Call Interface) Programmer's Guide, 10g Release 2 (10.2). ORACLE PRESS, OSBORNE, December 2005, B14294-02, 474p.

[6] SWIG, http://www.swig.org/.

[7] P. Mato et al. GAUDI: Architecture Design Document. LHCb 98-064 COMP.

Chapter 4 Requirements and use cases

This chapter presents the approach adopted for configuring the detector. First, we explain the methodology applied to build the 3 Tier architecture mentioned in the previous chapter. Secondly we present the requirements. Thirdly we describe the use cases collected in the frame of this project. Collecting requirements and use cases are steps which can be retrieved in most projects. The aim is to make an efficient and complete table design which should represent the information needed to configure the detector into the LHCb CIC DB. It also gives the outline of the tools which need to be implemented, namely it helps in developing a complete API.

4.1 Methodology

4.1.1 Identifying the users of the CIC DB

The CIC DB will be accessed by different types of users for different purposes. It is essential to know what kinds of requests these users will perform. Thus, all the users of the CIC DB must be identified to ensure that the list of requirements and use cases is complete.

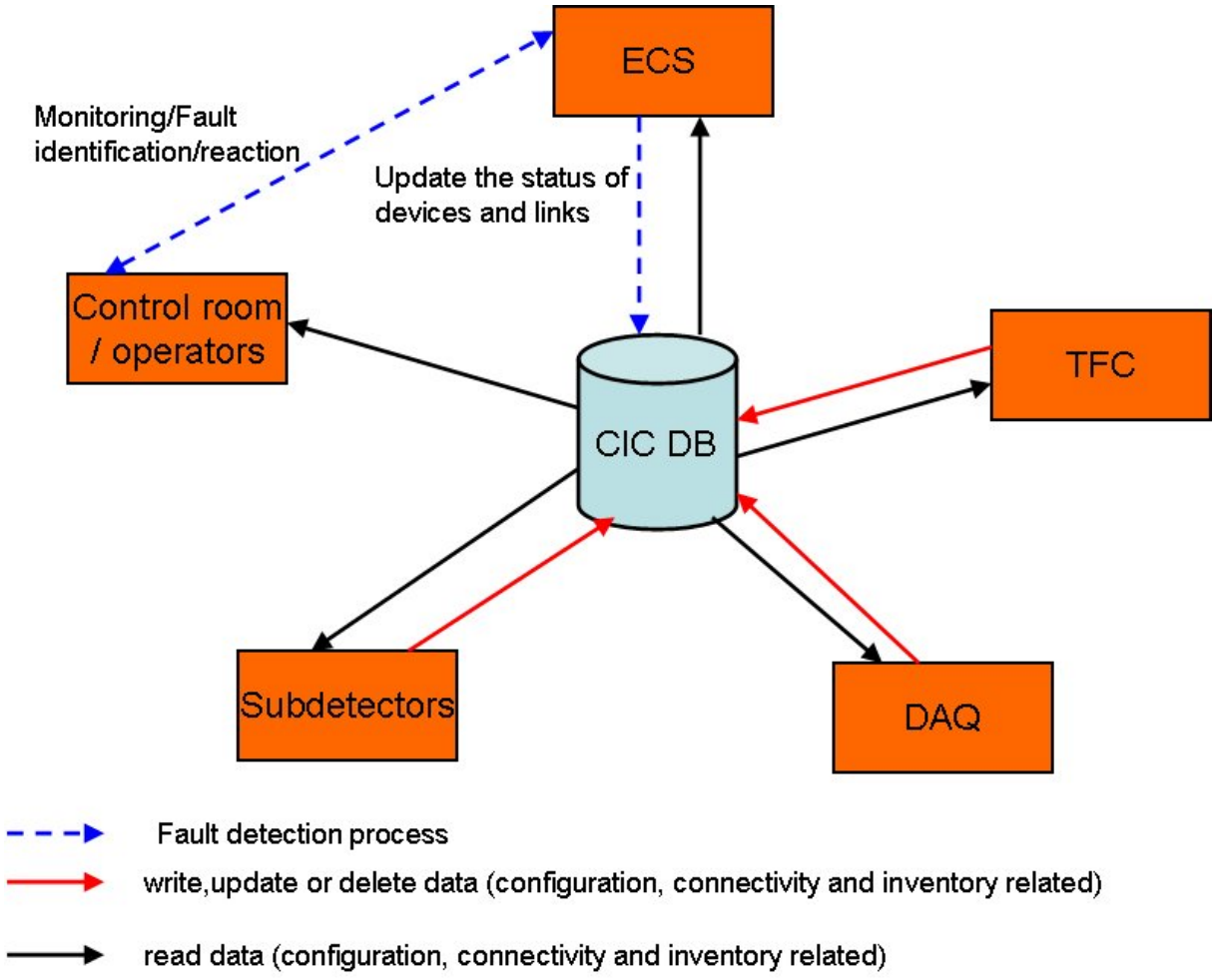


Figure 47. The different users of the CIC DB.

Figure 47 shows the different users which will interact with the CIC DB.

The subdetectors, the TFC and the DAQ need to save the different configurations of their electronics modules, their readout supervisors and farm PCs. These three groups of users should also be able to load configurations.

The TFC, DAQ and the subdetectors need to save the connectivity of their system to configure the TFC switch, to configure the network equipment and to configure some electronics modules (lookup tables for instance) and to test links. And on the opposite, they should be able to get the information (routing tables, DHCP config file, destination tables for the TFC switch, etc.).

The TFC, DAQ and the subdetectors need to save and query information about inventory and history of devices.

The ECS needs to mix the configuration, connectivity and inventory information. It needs to identify the devices in a partition, and to find out what configuration parameters with their values to apply according to the activity. The ECS will also query some connectivity information and will update information about the status of a device or a link. An update of the status of a device may occur further to a fault detected by the operators in the control room via some other tools. The aim of the ECS is to provide a configuration with a single-click by combining different types of information and a right reaction in case of failures in the detector. That is why the ECS has to be implemented using autonomics tools.

To permit the different users to interact with the CIC DB, it is also essential to know which tools/languages they will use to perform their requests. Table 5 shows the results we came up with (to access databases in LHCb Online environment, C, Python and the PVSS scripting language are the most common languages).

Users/tools	**C code**	**Python code**	**PVSS scripts**
DAQ	Yes	Yes	Yes
TFC	No	Yes	Yes
Detector	Yes	Yes	Yes
ECS	Yes	No	Yes

Table 5. The list of the CIC DB users and tools/languages used.

4.1.2 Listing the requirements

There are 4 types of requirements to define:

- The common and user specific requirements;
- The requirements for the integration of the project;
- The requirements for the user interfaces;
- The security requirements.

4.1.3 Collecting use cases

In the context of this thesis, a use case or scenario describes how users or applications will interact with the system. Our definition of use case is not exactly the same as the one used in UML [1].

Use cases are very helpful to model the information, i.e., configuration, connectivity and history data. They will answer the following questions:

- What are the entities and their attributes?
- What the relationship between entities?

- Which is the necessary data for configuration, connectivity and history?

By stepping through a use case, one can check whether the implementation of the system corresponds to the problems it was supposed to solve. Writing down use cases ensures that the system to design will be complete.

The next sections will describe the use cases which have been used to model the information for the database.

4.1.4 Understanding the LHCb environment

There are different types of software and electronics technology which are used. They are summarized in Table 6 (derived from Chapter 2). They have to be understood to make a an efficient design and to implement good tools.

Subsystem	**Technologies (electronics modules + software)**
ECS	PVSS and its frameworks (FSM, DIM, etc.), CCPC, SPECS, etc.
DAQ	Gigabit Ethernet, routers, HLT application, DHCP and DNS servers
TFC	Readout supervisors, TFC switch, fan-outs, Throttle switches.
Subdetectors	FPGA, HPDs, R- and Phy- sensors, PMT, and other boards etc.
Offline	BOOST in Gaudi, CVS, CMT

Table 6. Example of electronics modules and software used in LHCb.

4.1.5 Building the CIC DB table schema

The table schema has been built by analyzing the use cases. For each use case, we have modeled them with ERM diagrams. Some entities appeared in all use cases. Entity attributes have been defined in such way, that whatever the use case is, all the information can be put in one of the attributes. In other words, we have listed and grouped the different attributes that an entity can have.

In some cases, we had to design specific entities for a particular subdetector (essentially for the DAQ system with the different attributes specific to a switch). Unlike other subdetectors, the DAQ is a Gigabit network so it requires special settings which will be explained in section 4.3.2. However we tried to minimize the number of specific entities for software maintenance reasons.

4.1.6 Integrating and developing tools

The users will not type in directly SQL statements to interact with the CIC DB. Some interfaces will need to be developed so that the users can access and manipulate the information stored in the CIC DB. It will be achieved by defining a complete API to allow interacting with the CIC DB without any knowledge of its table schema. The deployment of these tools (based on the API) should satisfy the requirements about integration, security and user interfaces.

4.1.7 Test and validation

The table schema must be validated and all the tools and libraries developed to interact with the CIC DB must be tested. This includes a check of functionalities by looking back at the use cases and behavior in case of misuse.

4.2 Requirements

Modeling information in a database and implementing the tools depend a lot on the system requirements. Some of them (see list below) can be applied to any database schema [2].

4.2.1 Common and user specific requirements for the table schema

4.2.1.1 Completeness and consistency

The table schema must be complete and consistent. The LHCb ECS must be able to retrieve any information related to detector partitioning, device configuration, connectivity and device history. Also the information should be stored in a coherent manner by avoiding data redundancy as much as possible.

Thus it is necessary to specify which type of data should be stored in the CIC DB to satisfy these requirements.

4.2.1.2 Performance

Techniques to improve performance in reading (resp. in writing) from (resp. in) a database are different. Thus it is important to determine whether the database will be mainly read, mainly written or both.

Information retrieval from the CIC DB must be quickly executed. The CIC DB will be used most of the time in read-only mode as device configurations and connectivity information will be rather static once the detector will be in the operational phase. For instance, at start up, around 500,000 devices will have to be configured in 5-10 minutes. To reconfigure the detector (stopping a run, changing the activity or the partition of the run and starting the detector again), the 500,000 devices must be configured in less than one minute. It should take less time as the electronics modules are already powered up. It is similar to stop/start and log off/log in a computer. The second operation takes less time.

It is important to identify the most frequent requests from the CIC DB to be able to improve performance. It will also help in building indexes on tables.

It is important to have an idea of the number of simultaneous accesses to the database. It will have an impact on the physical design of the database (use of RAC –Real Application Cluster- [3] for instance) and on creating views in the schema. If there are many users which access the database concurrently, there is a need to provide a high availability. RAC is an Oracle technology which aims at ensuring availability using a distributed database system. A view is a selection of the content of one or more tables which fulfill certain conditions.

4.2.1.3 Extensibility

The table schema must be extendable. Requirements may change at a later time. For instance, for the moment there is no version implemented for connectivity. It means that it is always the

current connectivity which is stored in the database. If in the future, versioning needs to be introduced, it should not affect dramatically the table schema. In other words, it should be possible to easily extend the table schema.

4.2.1.4 Generic schema

Besides the rather general requirements listed above, the table schema must be applicable to all subdetectors. It means that the representation of the configuration, the connectivity and the history must cater for all subdetectors despite different types of electronics and different kinds of connectivity. In other words, the table schema needs to be independent of the subdetector. This way, one avoids having a table schema per subsystem. The maintenance of the table schema becomes easier.

4.2.2 Requirements for integration in the LHCb environment

We came up with the following list, after discussing with the LHCb collaboration.

REQ 1: Most of the users are not familiar with databases. There is a need to develop tools which allow manipulating the data stored in the CIC DB without typing any SQL statements.

REQ 2: As the configuration of electronics modules should be done from PVSS, an interface to PVSS must be implemented allowing access to the data stored in the CIC DB.

REQ 3: A tool to navigate in the LHCb CIC DB and to view the connectivity needs to be built. PVSS is not really meant for displaying the connectivity between devices in an automated and dynamic way as PVSS is not object-oriented. One of the requirements for this tool is to provide a function which gets all the paths going through a given device and a given subsystem. So there is a need to store to which subsystem(s) a device belongs.

REQ 4: Tools must be as autonomic as possible, especially at the level of equipment management. Human intervention should be minimized. For instance, the creation of routing tables must be automated and intelligent, i.e. if a links breaks, another routing path should be proposed.

REQ 5: An API must be developed allowing manipulation (query, insertion, update and delete) of configuration, connectivity and inventory/history information. The bindings of this API can be used by the tools mentioned in the requirements above.

4.2.3 User interface requirements

REQ 6: User-friendly;

REQ 7: Clear and simple so that it can be used without the intervention of the developer;

REQ 8: Complete by providing the required functionalities.

4.2.4 Security requirements

The LHCb experiment will be installed underground in a cavern. There will be no possibilities to access the devices of the experiment from outside once the detector starts to operate. The

computer network will be disconnected from the CERN Intranet. However some authorized gateway PCs will be accessible from outside as they will have an interface on the CERN network, in addition to an interface to the LHCb network.

The CIC DB will be installed using Oracle RAC. Back ups will be performed using Oracle backup features. It will be accessible only from the LHC b network so there are no security requirements.

4.3 Use Cases

In this section, we list the different use cases for configuration, connectivity and inventory/history.

Some use cases can appear more than once (they can be used for different purposes).

4.3.1 Recipes

A list of use cases [4] has been written down and given to the CERN PVSS support group as they are responsible for providing PVSS tools to configure the devices for the LHC experiments. This group has introduced the concept of recipes meaning a configuration for one of a set of devices. I took part actively in building the LHCb list of use cases by testing and evaluating the first prototype. One of the missing features was the possibility to save a partial recipe, i.e. being able to select some parameters of a device. We list the most important ones (all these use cases are for PVSS) needed by LHCb:

UC 1: The RICH group has received HPDs. They have started to design the configuration PHYSICS for the HPD device type. They save this configuration in the CIC DB on October, 22nd 2005 for half of the HPDs. Then they load the configuration PHYSICS and apply the setting to the half of HPDs it was saved for. They change one parameter and save this configuration again (same name and same devices) on October, 26th 2005. And they load the first version (from the 22nd) of the configuration.

UC 2: The VELO group saves a recipe COSMICS for the 88 hybrids (March, 15th 2006). They load it and apply to it to the devices. They change a few parameters. They save it as COSMICS_TEST for 44 hybrids.

UC 3: The ECAL group has started to design its local control system using FSM. They save the current FSM hierarchy (January, 14th 2006). On January; 28th they load this version on another PC. Then they design and save a recipe CALIBRATIONS for the whole FSM hierarchy (still January; 28th). They load the recipe CALIBRATIONS for the whole hierarchy (parameters + values + list of devices part of it).

UC 4: The ECS saves a recipe TEST for the devices part of the whole FSM tree. Then it loads the recipe TEST for a FSM subtree, and the recipe TEST is applied only to the devices of the subtree.

UC 5: The IT group has saved a lot of recipes for their devices. They do not remember exactly what the names are. They look at the list of recipes they already saved.

UC 6: The HCAL group has installed their equipment and start to design their local control system. Each channel will be illuminated by two LEDs. For calibrations purposes, they need to get which LED(s) illuminate the given channel name.

UC 7: The HCAL group has attributed a quantity of light which is used for computations for links between a channel and a LED. They get the coefficient value for one specific link.

An analysis of these use cases shows that the following requests will be performed against the CIC DB, from and only from PVSS:

- Save a recipe for one or several devices;
- Save a recipe for a hierarchy of devices;
- Save a hierarchy of devices;
- Save different versions of a recipe;
- Load a recipe of a given version for the devices it was saved for;
- Load a recipe (by default, the last version) for a subtree or a list of devices;
- Load a hierarchy of devices;
- Get the list of recipes for a given subsystem;
- Get the connectivity between two devices;
- Attribute a coefficient to a link;
- Get the light coefficient for a given link (specific to the Calorimeters subdetector).

4.3.2 Networking

In this subsection, we describe the use cases related to the DAQ network configuration.

UC 8: The DAQ group has set up their controls and data networks. They want to have static routing tables for performances and debugging reasons. They program the routing tables of the hundreds of routers by selecting the shortest path. They also configure the speed and the phy parameter (type of cable used, three possible values { "SX","T","SL"}) of their switches.

UC 9: The DAQ group configures their DHCP and DNS servers according to their set up.

UC 10: One PC in the DAQ farm is down. They want to get information about the device type (number of interfaces, description). They also disable the PC and update the routing tables for the routers affected by this failure. They also update the DHCP config file and the DNS files.

UC 11: Extra PCs have been added in the farm. For all of them, they enable the promiscuous mode and set which interface is used for booting, also called pxi_booting port. They update the routing tables, the dhcp config file and the DNS files.

UC 12: The DAQ group finds that a TELL1 board behaves strangely. They get some information about this board and its interfaces such as name, location, IP and MAC addresses, serial number, bia (burnt internal address), IP aliases. They also want to disable its two data network interfaces.

UC 13: The DAQ network includes several DHCP servers. They do not remember their location. They get the list of DHCP server names and their respective location. Then they

disable three links between switch A and switch B and update the routing table and the DHCP config files, because they want to make some measurement performance.

UC 14: The DAQ group gets the list of IP aliases for a given PC in the farm.

UC 15: Part of the DAQ equipment is used to extend cables (patch panels for instance). They do not want to store it as equipment but they need to know if a link is one long cable or several short cables connected via patch panels.

UC 16: The DAQ group has implemented a Flower topology in 2003. In 2005, the price of switches and routers became cheaper. They could afford to buy the Force Ten router. They defined a new topology. They want to only keep the current design in the database.

For the use cases **UC 8** to **UC 12**, the DAQ group wants to use autonomic tools as it is quite tedious and bound to mistakes to do all these operations manually.

From these use cases, the following requests are derived:

- Generate the routing tables for the DAQ routers;
- Generate the DHCP config file;
- Generate the DNS files;
- Update the routing tables when there is a change in the set up (new PCs added, a PC breaks);
- Update the DHCP config file;
- Update the DNS files;
- Get information about a device and its interfaces;
- Get the list of IP aliases for a given device;
- Get the list of devices which fulfills a specific task (DHCP, DNS server);
- Insert the connectivity of the system;
- Add information about a link;
- Delete the connectivity of the system.

4.3.3 Partitioning

The partitioning affects the TFC system. The three following use cases have been collected.

UC 17: The shift operator has defined the following partition {ST, VELO, L0 TRIGGER, ECAL}. The TFC switch is configured accordingly.

UC 18: The TFC group has received 2 extra readout supervisors. They get the current list of readout supervisors. They check to which inputs of the TFC switch they are connected. They install and connect the two extra ones.

UC 19: The shift operator selects a region of one station in MUON subsystem as partition for debugging reasons. The TFC sends the clock to this group of devices.

The requests which come up are as follows:

- Program the TFC switch according to the partition, i.e. to which outputs of the TFC switch these subsystems {ST, VELO, L0 TRIGGER, ECAL} are connected;
- Get the list of devices of a given type;
- Get the input or output connectivity of a device;
- Get the output port of the TFC switch which sends signal to a group of electronics module.

4.3.4 Equipment management

4.3.4.1 Scenarios

This subsection gives a list of the use cases related to equipment management:

UC 20: The VELO group has received 30 spare hybrids. They named them from spare_hybrid_01 to spare_hyrbid_30. They are stored in building 15, R-008, on August, 23rd 2007. On January, 2nd, they make an inventory of all their spares. And how many do they have per type?

UC 21: The OT group has sent all the devices of type OT_C_TELL1 to be repaired these two last months. However, they do not remember their names. They ask for the list of devices with the name of the responsible which are being repaired.

UC 22: The Silicon Tracker group has noticed that one of its TELL1 boards often fails this year. They get the history (of this device) which describes the different statuses with comments and the date of this change of status, and the location (where it has been repaired for instance) for this year. They want to know how many spares of type TELL1 there are in their laboratory.

UC 23: MUON_TELL1_12 has been used to take data since July, 21 2006. Then it broke on October, 23 2008 and has been sent to be repaired. MUON_TELL1_12 has been replaced on October, 24 2008.

UC 24: The HCAL has 5 PMTs which are destroyed. So they want to store this information. They also want to get the status of HCAL_PMT_05.

UC 25: The ECAL has 10 DAC boards which were working fine until today. They need to send these boards to Orsay for some tests. It should be known that these devices are in Oxford for some tests. One month after, the RICH 1 group wants to get the status of the HPD identified by XX67HGJK90.

UC 26: A readout supervisor is installed (underground) and is working. Two days later, it has a slight problem which can be fixed quickly. It is taken out from its emplacement and put in a special test area where it can still have an IP address part of the internal network. There is a specific number of boards which can be tested locally (still in the cavern). It should be reported that this readout supervisor has to be taken out for local tests.

UC 27: The MUON group has a lot of Muon Front-End electronics which need to be installed. Unfortunately, they made a mistake when reporting their location. They put half of

them in station 1 and the others in station 2, whereas it should have been the other way around. They must be able to correct this error.

UC 28: A chip located on a RICH L0 electronics board fails. It has been replaced on its own by another one in spare if any (of the same type). They need to connect it as it was before. They also need to verify that all the components of the board have been replaced.

UC 29: A chip located on a PRS DAC board does not work. The whole board needs to be replaced. One week after, a new board comes but the PRS group does not know where exactly it was. They need also to verify whether the chip intel_chip_08 has been repaired or not. If yes they need to put it back on the motherboard. If not, they need to ask the person responsible for it to know the issues.

UC 30: The RICH group wants to get the history of the chip RICH_CP_12 located on a L0 electronics module to verify how many times it fails. They also want to get the list of devices (board components included) which have not been replaced yet.

4.3.4.2 Device status

So the possible statuses (derived from the use cases) that a device (included board components) can have are the following:

- IN_USE (UC 23). It means that the device is installed in the cavern (underground) with a specific location and is ready to be used.
- SPARE (UC 28). It means that the device is on the shelf and it is not connected. It will be used to replace a device.
- IN_REPAIR (UC 23). It means that the device is in reparation further to a failure or a break down.
- DESTROYED (UC 24). This is the worst case. The device is dead.
- EXT_TEST (UC 25). It means that the device has been taken out from the pit and it is tested in a lab (inside or outside CERN). Usually the scenario is the following. The device is IN_USE and then it behaves badly. So it is taken out from its place to be tested elsewhere.
- TEST (UC 26). It means that the device is tested locally, still in the pit. So contrary to EXT_TEST, the device will benefit from the local infrastructure (i.e., a location, IP and MAC addresses if necessary). The scenario is the following. The device is IN_USE and then it gives strange results. It will be taken out from its location and tested locally in a place reserved for local tests. Usually it is used when the expert thinks that it can be fixed quickly. If it is more serious than foreseen, the device is tested in a lab with more tools. In that case, the device changes its status and goes from TEST to EXT_TEST.

When the status of a device changes its new location and the date of change must be reported.

For replaceable microscopic components, as not all of them can be replaced, the different statuses are the same as for the devices. If they are not replaceable, their status corresponds to the one of the motherboard.

4.3.4.3 Allowed transitions

In my case, Table 7 shows the allowed transition, given the initial status of a device.

Device initial Status	Is allowed to go to status
IN_USE	SPARE, IN_REPAIR, TEST, DESTROYED, EXT_TEST
SPARE	IN_USE
TEST	SPARE, IN_REPAIR, IN_USE, DESTROYED
EXT_TEST	SPARE, IN_REPAIR, IN_USE, DESTROYED
IN_REPAIR	SPARE, IN_USE, DESTROYED
DESTROYED	

Table 7. Allowed transitions.

Same rules can be applied for replaceable microscopic components.

4.3.4.4 Duality between hardware and functional devices

UC 23 raises the following problem:

Is MUON_TELL1_12 before October, 23 2008 the same as after October, 23 2008?

From a hardware point of view, the answer is no whereas from a functional point of view the answer is yes. Indeed MUON_TELL1_12 as functional device will perform the same functions before and after October, 23 2008. However the hardware device which occupies the MUON_TELL1_12 function is not the same before and after October, 23 2008.

So there are two dimensions to take into account, that are functional and hardware.

A hardware device is uniquely identified by a serial code. A functional device is uniquely defined by its functional name. If a hardware device is IN_USE, it will inherit the location of the functional device it occupies. The location is associated with the functional device when the hardware is IN_USE.

Using this concept the previous scenario is reformulated as follows. MUON_TELL1_12 associated with the hardware device XDG67FDG77 has been IN_USE since July, 21 2006. Then it broke on October, 23 2008. MUON_TELL1_12 has been replaced by the spare GHOFD89878 on October, 24 2008.

The status of the functional device is deduced from the status of the hardware device associated with this function as it is shown in Table 8. Table 8 should be read as follows if a hardware device is IN_USE, then the functional device associated is IN_USE.

The status of a functional device can be {IN_USE, NONE}. The status NONE means that there is no hardware device which can perform the function.

The same remark can be made for configuration and connectivity. They are also functional concepts. Devices will be modelled in PVSS using functional devices, otherwise whenever a hardware device is replaced or destroyed, it implies to update all the device names which is clumsy. For the connectivity, it is the same thing. Switches, DHCP and DNS servers use names and not serial numbers. Also for the partitioning, it is a list of functional devices which will be given, not a list of serial number. Moreover it is more meaningful for the users to talk about MUON_TELL1_12 than GHOFD89878.

Hardware device status	Functional device status
IN_USE	IN_USE
TEST	NONE
EXT_TEST	NONE
IN_REPAIR	NONE
SPARE	NONE
DESTROYED	NONE

Table 8. Correspondence between hardware and functional device statuses.

The duality between hardware and functional devices is also applicable to microscopic devices.

4.3.4.5 Queries

The following types of queries will be performed according to the use cases.

- Get the list of spare devices (hardware devices) given a location;
- Get the list of spare microscopic devices (hardware components) given a location;
- Get the list of spare microscopic devices (hardware components) given a type;
- Get the list of spares of a given type;
- Get the history of a device (either hardware or functional) over a certain period;
- Get the history of a microscopic device (functional or hardware);
- Get the list of devices (hardware or functional) which have the given status;
- Get the list of microscopic devices (hardware or functional) which have the given status;
- Replace a functional microscopic device with a spare;
- Replace a functional device with a spare;
- Update the status of a device (either functional or hardware);
- Update the status of a microscopic device (either functional or hardware);
- Update information about a microscopic device (functional or hardware) due to mistypes;
- Update information about a device (functional or hardware) due to mistypes.

4.3.5 Fault detection

The following use cases describe different ways to detect fault in the experiment.

UC 31: The OT subsystem installs their equipment. They test their chain of equipments by injecting patterns, from device A to device B. Device B does not receive any signals. Where and what is the faulty device or link?

UC 32: A VELO hybrid consists of 16 beetles chips. They want to get the names of them stored on the hybrid_34. Each group of 4 sends signal to a mezzanine driver located on a repeater board. One beetle chip located on a VELO hybrid fails. Which mezzanine driver on which repeater board is affected?

UC 33: Some TT TELL1 boards give strange results. They are installed in the same rack. Where is the problem? Is it the rack or the TELL1 boards? They swap one of these TELL1 boards with a functional one. They compare the result. It seems that it is due to the rack. They redo the swap. They get the history of this rack to check if there were any previous failures.

UC 34: A RICH TELL1 XX board does not work properly during data taking. It is replaced by another one, if there is one. And XX is tested locally benefiting from the underground infrastructure (IP addresses, etc.). It still does not work. It goes back to the institute who built it.

These use cases imply the following requests:

- Get the paths between a functional device A and a functional device B;
- Get the paths between a microscopic component C and a microscopic component B located on two different motherboards;
- Swap two functional devices;
- Update the status of a functional or a hardware device.

4.4 Conclusion

In this chapter, we have seen the methodology applied to design the CIC DB table schema and the appropriate tools which interact with the CIC DB. The approach includes identifying the groups of users and collecting the different requirements and use cases. One of the difficulties is to make sure that the list of use cases and requirements is complete and correct.

In the next chapter, the design of the table schema is described using the use cases defined in this chapter.

References

[1] Alistair Cockburn, Writing Effective Use Cases, Addison Wesley Professional, Otcober 2000. ISBN10: 0201702258. 304 p.

[2] L. Abadie, E.v.Herwijnen, C. Gaspar, R. Jacobsson, B. Jost, N. Neufeld, the LHCb configuration database. In the proceedings of ICALEPCS 2005, Geneva, 10-14 Oct. 2005. MO4A.2-7O.

[3] ORACLE, Oracle® Database, Oracle Clusterware and Oracle Real Application Clusters Administration and Deployment Guide, 10g Release 2 (10.2). January 2006. B14197-03. 394 p.

[4] List of requirements and use cases regarding the PVSS configuration framework for LHCb http://lhcb-online.web.cern.ch/lhcb-online/configurationdb/default.htm#Doc

Chapter 5 The LHCb CIC DB schema

This chapter describes the table schema part of the database layer. First we explain why the ERM [1] has been chosen to design the table schema. We also give a brief overview of the steps performed to build the table schema. Secondly we give an overview of the ERM model. Thirdly we explain how the recipes have been represented in the CIC DB. Then we describe the representation of the inventory/history information which has aroused the problem of the duality between hardware and functional devices. Then the connectivity representation (for the two levels microscopic and macroscopic) is shown. We also show how the N:M relationship has been replaced with a method based on prime numbers. Finally we verify that the table schema is complete using the use cases defined in the previous chapter.

5.1 Introduction

5.1.1 Why the ERM?

Besides the ERM, there are different ways of representing data such as object databases.

Object databases are very useful when the applications accessing the database are written in an object language. They are also very convenient as there is no need to know SQL to retrieve or store data. However, an object database is directly linked to the object format and data types of the host language. This causes problems when other languages attempt to query data from the store. So an object database is not as flexible as a relational database.

Relational databases are complex to design as there are different ways to model the same information using the ERM. But they are reliable, and they ensure data integrity.

In LHCb online context, the use cases have been analyzed to see if there is any need to model data as an object. In fact, I concluded that all the data could be modelled using conventional types (number, varchar2) that can be stored in a column. There is no need to create special types which could necessitate a design using objects.

5.1.2 Designing the table schema

As I used the ERM, the methodology to build the table schema is standard and as follows:

- Collect and analyze use cases.
- Apply the ERM to each use case. Determine entities with their attributes and the relationship between entities. Draw ERM diagrams.
- Integrate all the ERM diagrams generated from the list of use cases to avoid duplication of information and identify missing information.
- Submit the ERM diagrams to the different users to make sure that the information is complete.
- Define a primary key [1] for each entity and foreign keys [1] to express the relationship between entities. Build the table schema.
- Improve the performance of the database wherever possible.

5.1.3 Conventions

- Entities and table names are written in bold **UPPERCASE**. In Figure 56, Figure 64, Figure 76 and Figure 77 the table names are written in UPPERCASE.
- Attributes and columns are written in bold **lowercase**. In Figure 56, Figure 64, Figure 76 and Figure 77 the columns are written in lowercase.
- "pk" means primary key and "fk" means foreign key.
- "I" means index and U means unique index with a unique constraint.

5.2. Entity Relationship Model (ERM)

This section gives the principles of the ERM.

5.2.1 Entity

An entity type is similar to an object oriented class. It groups a set of similar elements. For instance, in the CIC DB, an example of an entity type is a device, as it has a number of characteristics and a device can be uniquely identified using its serial code. An entity can be concrete as a person or a device or it can be abstract as history or a concept.

5.2.2 Attributes

The characteristics of an entity are modelled as attributes such as (for the device entity example) functional name, location, name of the person responsible for this device, serial number or status.

The domain of the attribute specifies the allowed values. For instance the status must be one of the following predefined statuses ('IN_USE', 'EXT_TEST', 'TEST', 'IN_REPAIR', 'DESTROYED'). An RDBMS represents an attribute by a table column.

5.2.3 Relationships

A relationship is an association between several entities. Relationships express how entities are connected with each other. For example, the relationship "device_link" describes the association between the entity sets device and link**.** It is a binary relationship as there are two entities involved. A relationship can involve more than 2 entities.

In the ERM, a relationship has a cardinality. It sets the number of entities which are related to each other.

There are 4 possibilities:

1. **One-to-one**: an entity A is associated with at most one entity B, and an entity B is associated with at most one entity A. For instance, in the LHCb context, a boot image is associated with one device type. And vice-versa a device type has at most one boot image. There is no dual-boot situation as all the PCs in the farms are based on Linux.
2. **One-to-many**: an entity A is associated with any number of entities B. An entity B is associated with at most one entity A. For instance, let us consider the device type and device entities. Many devices can be of the same device type, but a device can be of only one device type.
3. **Many-to-one**: an entity A is associated with at most one entity B. An entity B is associated with any number of entities A. (This is the reverse of the previous case).

4. **Many-to-many**: entities A and B can be associated with any number of each other. For instance, a device can be part of different subsystems and a subsystem has several devices.
   The cardinality for a particular relationship depends on the data to model.

5.2.4 ERM diagrams

The Entity Relationship model is represented using diagrams as shown in Figure 48.

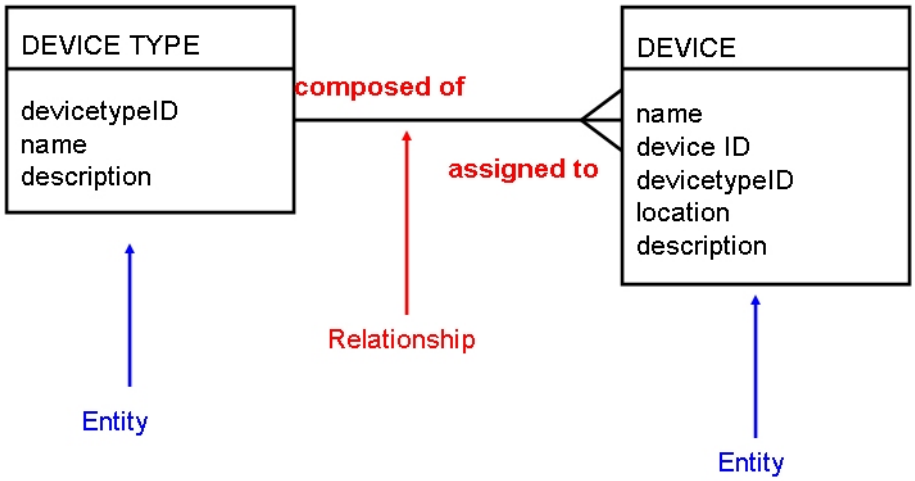


Figure 48. Diagrams showing entities, attributes and relationships.

Entities are represented using rectangles. Attributes of an entity are defined inside the lower partition of the entity rectangle. For instance, referring to Figure 48 **DEVICE TYPE** is an entity and **devicetypeID** is an attribute of the entity **DEVICE TYPE**.

Relationships are drawn differently according to the cardinality. A dashed line means that the relationship is optional.

- One-to-one relationships are drawn as shown in Figure 49 (attributes are suppressed here). A **BOOT IMAGE** will be used to boot one **DEVICE TYPE**. A **DEVICE TYPE** can have at most one **BOOT IMAGE** (dashed line towards **DEVICE TYPE** as not all the **DEVICE TYPEs** need a **BOOT IMAGE**).

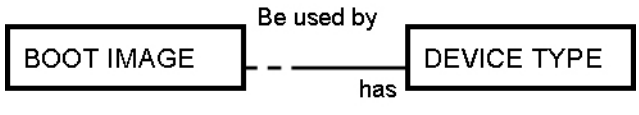


Figure 49. The drawing convention for one-to-one relationships.

- One-to-many relationships are drawn as shown in Figure 50. A **DEVICE** can have one or several **PORT**s. A **PORT** belongs to one **DEVICE**.

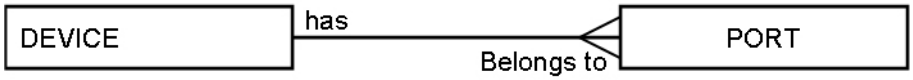


Figure 50. One-to-many relationship.

- Many-to-many relationships are drawn as shown in Figure 51. A **DEVICE** can be part of several **SUBSYSTEMs** and a **SUBSYSTEM** contains many **DEVICEs**

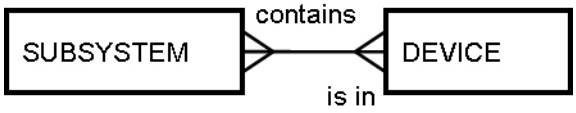


Figure 51. Many-to-many relationship.

5.3 From ERM to RM

The schema for the relational database (relational model) is derived from the ERM diagrams. The next subsections explain the mapping to perform to go from the ERM to the RM.

5.3.1 Tables

The relational model uses tables as a basic structure. An entity corresponds to a table, its attributes correspond to the columns of the tables and the domain corresponds to the data types. For instance, referring to Figure 48, the **DEVICE TYPE** entity is mapped to the RM as follows:

DEVICE TYPE <<Table>>		
devicetypeID (number) PK	name (varchar2) U	description (varchar2)
142	ODIN	readout supervisor

Figure 52. The DEVICE TYPE table. U stands for unique constraint.

Figure 52 shows the representation of the **DEVICE TYPE** entity in RM. The **DEVICE TYPE** is the name of the table. **DevicetypeID** (which is of type number[4]), **name** and **description** of (which are of type varchar2) are the columns of the table.

One row of the **DEVICE TYPE** table (also called tuple) represents an instance of the entity. (142, ODIN, readout supervisor) is a tuple corresponding to a particular **DEVICE TYPE** used by the TFC.

[4] Number and varchar2 are types used in Oracle databases. They denote integer values and character strings respectively

5.3.2 Keys

Relations between tables which model the association in the ERM diagrams are expressed with keys.

- A superkey is a set of one or more columns which allow a unique identification of a row in a table. For example, in the table **DEVICE TYPE**, **devicetypeID** is a superkey. A candidate key is a superkey that is minimal in the number of its columns.
- A primary key (PK) is a candidate key (there may be more than one) chosen by the DB designer to identify a row in a table.
- A unique key (U) is also a candidate key which could have been selected to identify a row in a table. A unique key (constraint) allow the DB designer to make sure that each value of the column(s) is unique (no repetition). For instance in the table **DEVICE TYPE**, there is a unique constraint on **name** as shown in Figure 53.

DEVICE TYPE <<Table>>		
devicetypeID (number) PK	name (varchar2) U	description (varchar2)
142	ODIN	readout supervisor

Figure 53. The DEVICE TYPE table with its keys.

- A foreign key is a column (or set of columns) of a table which refers to the primary key of another table. It enforces referential integrity. For instance, the column **devicetypeID** in the table **DEVICE** refers to the column **devicetypeID** in the table **DEVICE TYPE**. So the row identified by **devicetypeID** in table **DEVICE TYPE** cannot be deleted as long as a reference to the column **devicetypeID** exists in the **DEVICE** table. Foreign keys can be used to model 1: 1 or 1: N relationships.

  1:N relationships (see Figure 50) are modelled in the RM as shown in Figure 54. In Figure 54, and Figure 55, the dashed arrow indicates that there is a foreign key between the two columns. The arrow points to the column which is referred to. The primary key of the **DEVICE TYPE** table (**devicetypeID**) has been added as a foreign key column in the **DEVICE** table. In Figure 49, to map this relationship in the RM, a foreign key column **devicetypeID** has been added to the **BOOT IMAGE** table. To map a 1:1 relationship into the RM, one of two tables must contain a foreign key column which corresponds to the primary key of the other tables.

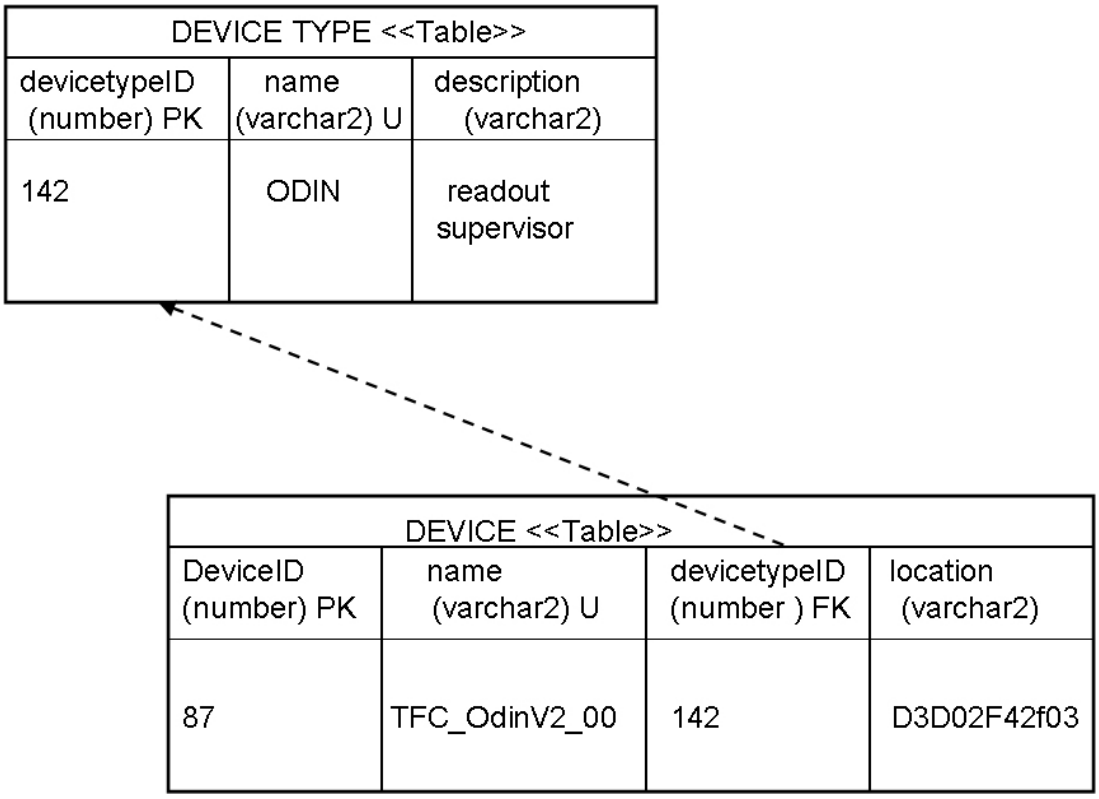


Figure 54. Representation of the 1:N relationship in the RM. The dashed arrow indicates that DEVICE.devicetypeID is a foreign key to DEVICE_TYPE.devicetypeID.

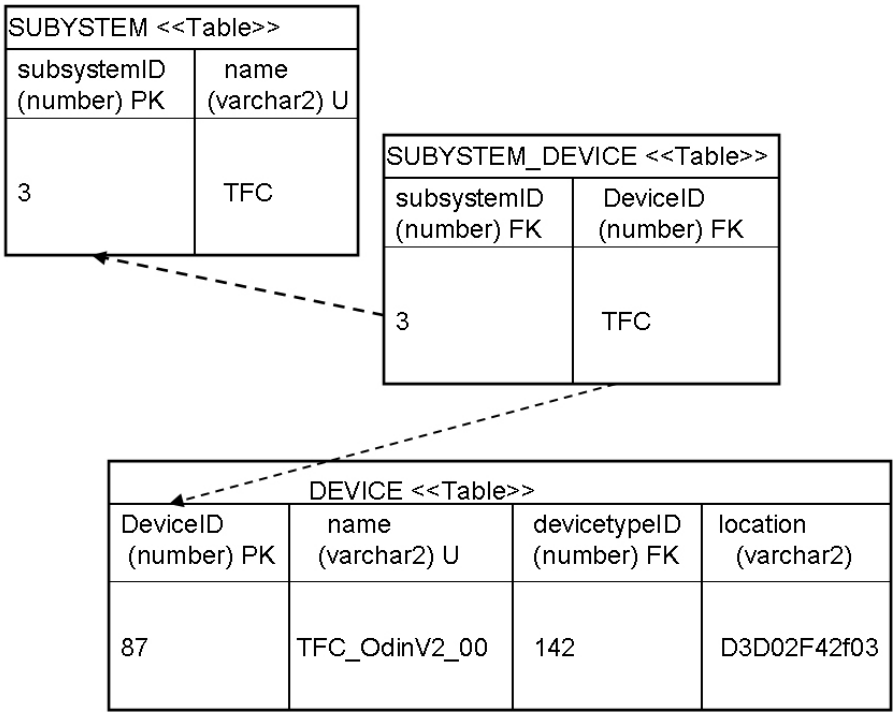


Figure 55. N:M relationship represented in the RM. The dashed arrow indicates that SUBSYSTEM_DEVICE.DeviceID is a foreign key to DEVICE.deviceID.

To model N:M relationship in the RM model, an extra table must be created. The primary key of the two tables must be added as foreign key columns in this extra table.

Figure 51 represents an example of N:M relationship. It is modeled in the RM as shown in Figure 55. The **SUBSYSTEM_DEVICE** table has been added to model the N:M relationship.

Two columns containing the primary keys of the two tables are mandatory. In the example used by the figures the columns are (**subsystemID**, **deviceID**) from the tables (**DEVICE** and **SUBSYSTEM**). The primary key of the **SUBSYSTEM_DEVICE** table is the pair (**subsystemID**, **deviceID**).

5.4 Recipe representation

5.4.1 Entity & relationship

To store the contents of the PVSS datapoints and the structures made by them (recipes), the CERN PVSS Support group have defined the following entities and attributes [2]. The entities and relationships have been defined based on use cases from UC 1 to UC 5 (in Chapter 4).To model a subsystem composed of a hierarchy of devices, two entities are used. A **HIERARCHY** has a **unique identifier**, a **type** (Hardware, Logical or FSM) and a description. An **ITEM** is a tree of devices. Each child has a single parent. An **ITEM** is associated with a **HIERARCHY**.

The **RECIPE** entity describes generic information of a configuration. It has a **name** and a **description**.

The **RECIPE DATA** entity describes the content of a recipe**.** It is a set of parameters and values. Each row of the table corresponds to a pair (parameter (**propname**), value (**propvalue**)) It has also **alert parameters** which are part of PVSS data points. For each parameter, there is the corresponding data type (**proptype**) as defined in PVSS. **RECIPE DATA** is associated with a **HIERARCHY** of devices.

The **RECIPE TAG** entity attributes a **tag** to each **RECIPE DATA**.

5.4.2 Representation with tables

The tables for storing recipes have been constructed as represented in Figure 56.
The **HIERARCHY** table contains the different types of hierarchies and their versions. Each hierarchy is uniquely identified by **hver**, the primary key of this table.

The **ITEMS** table contains the devices of the hierarchy identified by **hver** (foreign key). The **parent** column corresponds to the ID of the parent node in the hierarchy. For instance, the root of a tree has **parent**=NULL. The **name** column is the name of the device and **type** corresponds to the type of the device. For instance, if **name** is 'VEL0_TELL1_55', then **type** is 'VELO_TELL1'. **name** and **type** are the same as stored in PVSS data point and data point types. The **id** column is the primary key of the table, a sequence of number. The **dpid** is a foreign key to **id** column. This column indicates the parent device of **name**.

The **RECIPE** table lists all the different recipes which have been created. **Rver** is a sequence of numbers and is the primary key of the table.

The **RECIPE_DATA** table contains the collection of parameters of a recipe identified by the column **rver** and associated with a hierarchy node by the **id** column. The **propname** column contains the name of parameter to configure. The **propvalue** contains the value of this parameter. **Proptype** stores the type of the **propvalue**, i.e., if it is an int, a string or a bit, etc. it allows converting the data stored in the CIC DB in the correct PVSS types. The other columns are alarms used in PVSS. **Propid** (a sequence of number) is the primary key of the table.

The **RECIPE_TAG** table lists all the different recipe tags. The element parts of the recipe are listed using the **propid** column (foreign key referencing **RECIPE_DATA**.**propid** column).

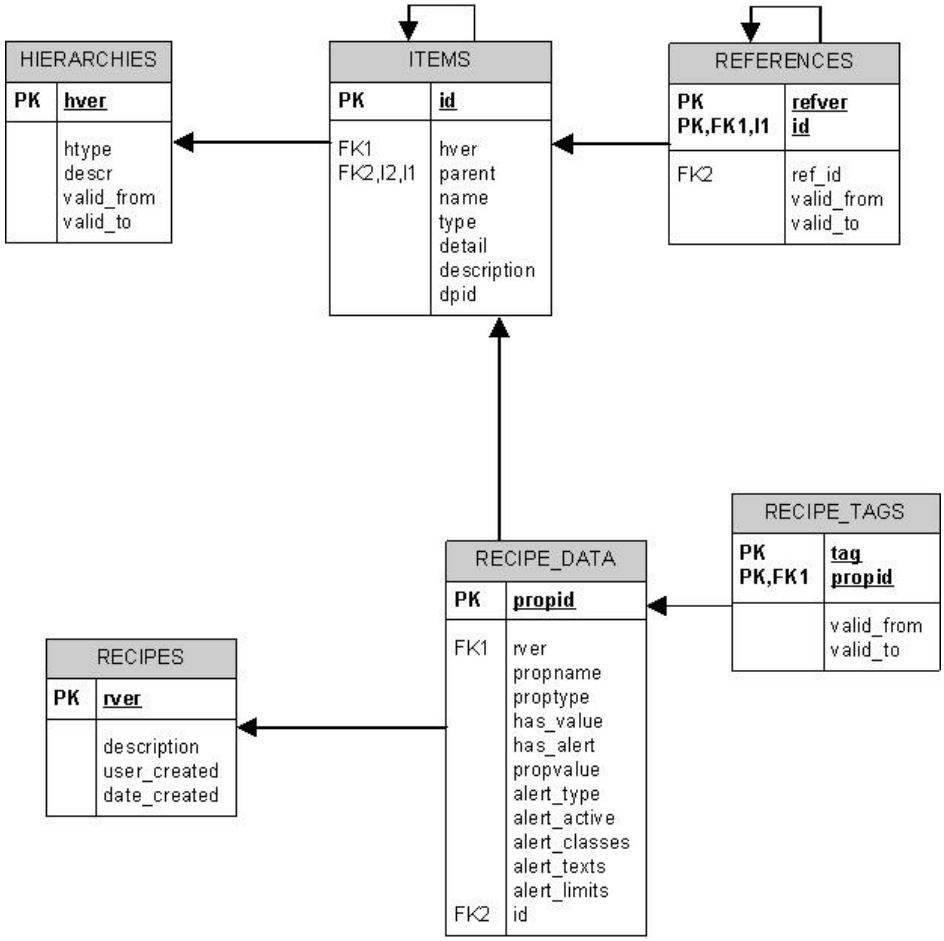


Figure 56. Table schema for the recipes

The **V_ITEMS** and **V_ITEM_NAMES** tables are materialized views which display the hierarchy structure in a better structure for PVSS.

5.5 Inventory and history design

5.5.1 Entity & relationship

Using UC 20, there are two entities to distinguish **HARDWARE DEVICE** and **FUNCTIONAL DEVICE.** Their respective attributes have been derived using different use cases defined in Chapter 2.

A **HARDWARE DEVICE** shown in Figure 57 is identified by a **serial code** (UC 25), an intrinsic property of hardware. It can have a **hardware (hw) name** (UC20). A hardware device has an **hw type** (UC 20), a **responsible** (UC 21) and a **current status** (UC 25). An hw device has a **location** (UC20).

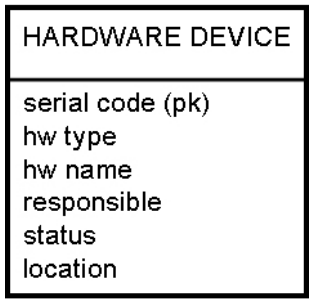


Figure 57. HARDWARE DEVICE model.

A **FUNCTIONAL DEVICE** as shown in Figure 58 has a unique **name** (UC 23) through all the experiment. It has also a functional **type** (UC 21). It can have a function (**functionID**) (UC 13). For instance, a controls PC can host both a DNS server and a DHCP server. A **FUNCTIONAL DEVICE** can be occupied by at most one **HARDWARE DEVICE** (UC 20). It is a one-to-one relationship. It has a **location** (UC 13). If the **FUNCTIONAL DEVICE** is occupied by a **HARDWARE DEVICE**, the **HARDWARE DEVICE** inherits from the functional **location**.

A **FUNCTIONAL DEVICE** can be **enabled** or **disabled** (UC 10), i.e. the **FUNCTIONAL DEVICE** takes data or does not take data. It is specified by **nodeused**.

A **FUNCTIONAL DEVICE** can have a **promiscuous mode** (UC11). This attribute is needed for the DAQ for PCs. This property can be seen *a priori* as a hardware property. However the value of this parameter is bound to the function. If the hardware is replaced, the value of this parameter will remain the same.

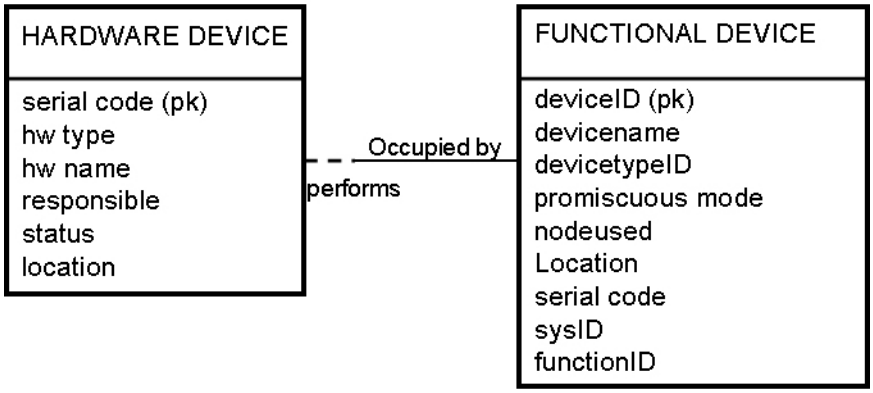


Figure 58. FUNCTIONAL DEVICE model.

A **FUNCTIONAL DEVICE TYPE** (UC 21) as shown in Figure 59**,** groups all the **FUNCTIONAL DEVICES** of the same type. There is a one-to-many relationship from **FUNCTIONAL DEVICE TYPE** to **FUNCTIONAL DEVICE**. A **FUNCTIONAL DEVICE TYPE** has a **name** (UC 21), **a number of inputs** (UC 10) and **a number of outputs** (UC 10). It has also a **colour** for display purposes (for CDBVis). **Input** and **output**

numbers are related to the **FUNCTIONAL DEVICE TYPE**. Indeed the number of **inputs** or **outputs** will not change if the hardware is replaced.

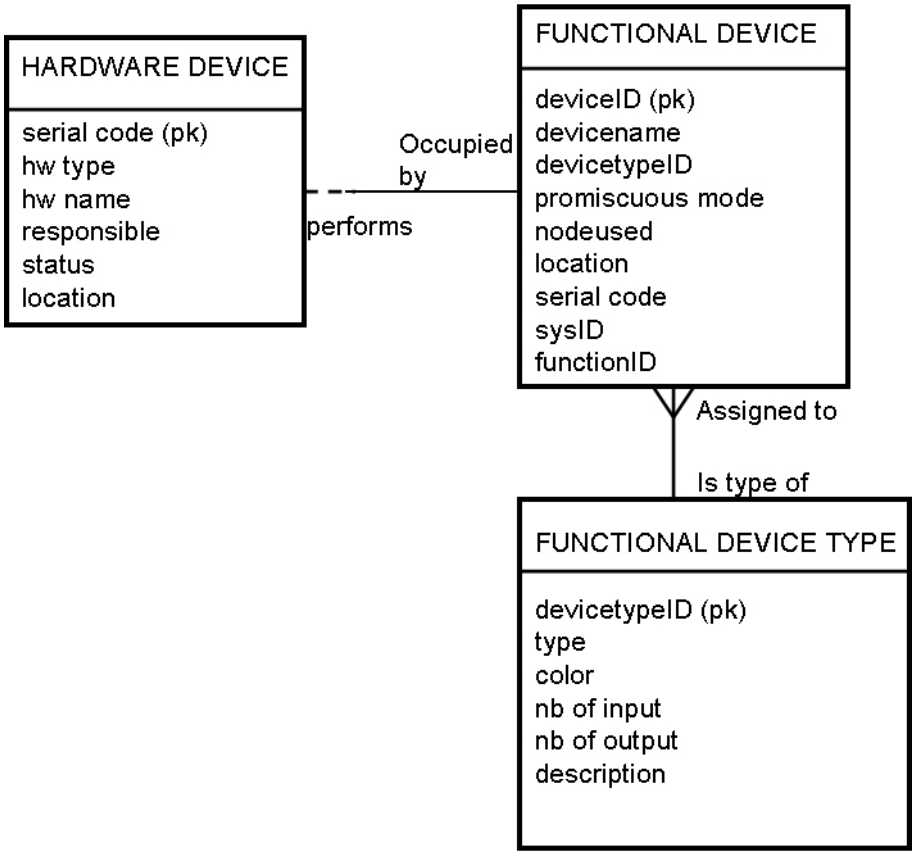


Figure 59. FUNCTIONAL DEVICE TYPE model.

The **HISTORY** (UC 30) of a given **HARDWARE DEVICE** or of a given **FUNCTIONAL DEVICE** consists of providing the following information (see Figure 60):

- **Serial code** (UC 20)
- **Deviceid** (functional device name is then derived) (UC 20)
- **Status** (UC 22)
- **Date of the status change** (UC 22)
- **Location** (UC 22)
- **Comments** (UC 22)

There is a one-to-many relationship from **HARDWARE DEVICE** to **HISTORY OF DEVICE** and also from **FUNCTIONAL DEVICE** to **HISTORY OF DEVICE.**

The **HISTORY OF DEVICE** table is common to **HARDWARE DEVICE** and **FUNCTIONAL DEVICE** tables.

FUNCTIONAL DEVICE and **HARDWARE DEVICE** are entities which are used in the macroscopic view. Two other entities have been designed to handle the microscopic view, **HARDWARE BOARD COMPONENT** and **FUNCTIONAL BOARD COMPONENT.** They present some similar attributes to the ones defined for the macroscopic view.

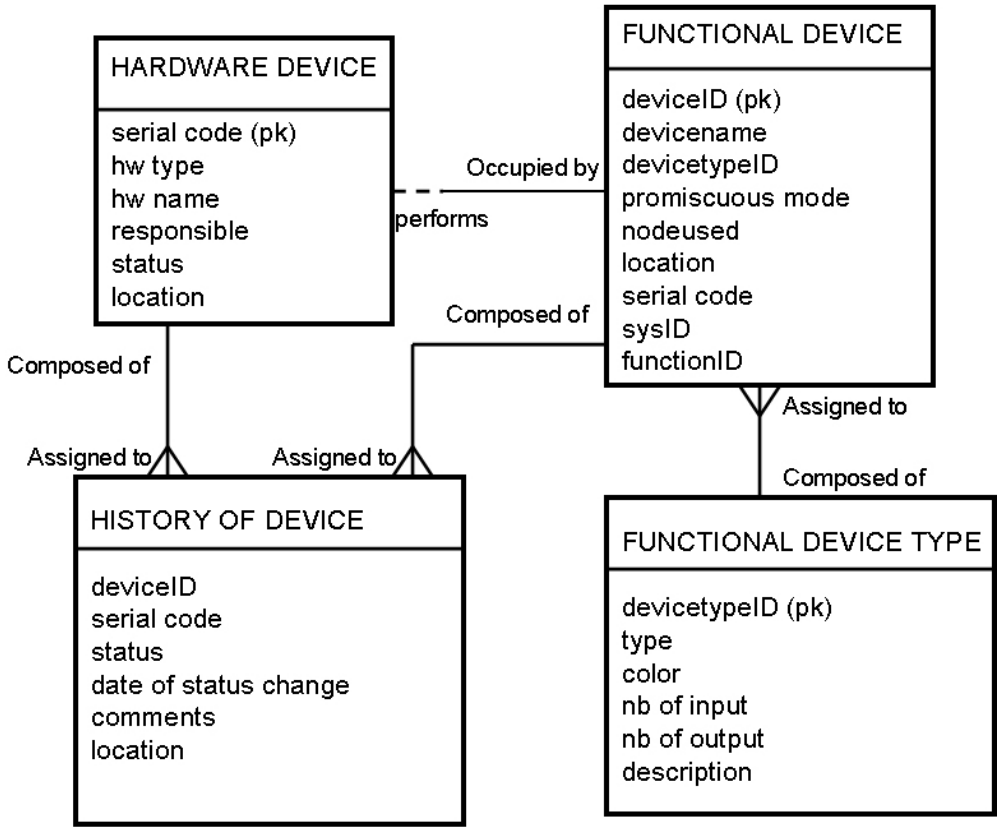


Figure 60. History model.

A **HARDWARE BOARD COMPONENT (**see Figure 61**)** (UC 28) has a **name** (UC 29), a **type**, a **responsible** (UC 29) and a **status** (UC 28). It can be **replaceable (**just the piece of hardware**)** (UC 28). If it is replaceable, the **HARDWARE BOARD COMPONENT** has its own **serial code** (UC 30).

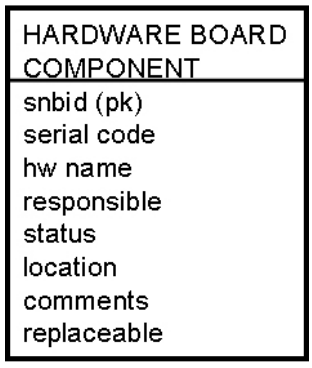


Figure 61. HARDWARE BOARD COMPONENT model.

If the **HARDWARE BOARD COMPONENT** is IN_USE or if the **HARDWARE BOARD COMPONENT** is not replaceable, then its **location** corresponds to the **HARDWARE DEVICE** where it sits.

A **FUNCTIONAL BOARD COMPONENT** (see Figure 62) has also **a name** (UC 30) and **a type** (UC 29). It is also occupied by at most one **HARDWARE BOARD COMPONENT**. It is a one-to-one relationship. The **location** of a **FUNCTIONAL BOARD COMPONENT** corresponds to the **FUNCTIONAL DEVICE** where it sits (UC 28). There is a many-to-one relationship from **FUNCTIONAL BOARD COMPONENT** to **FUNCTIONAL DEVICE.**

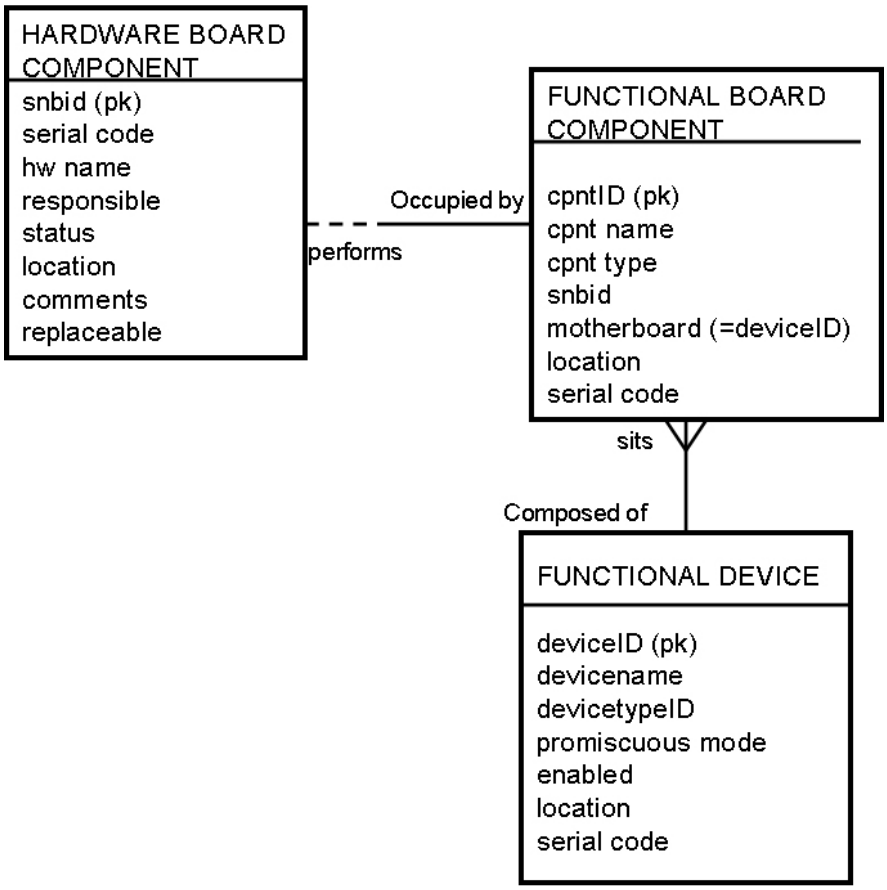


Figure 62. FUNCTIONAL BOARD COMPONENT.

A **BOARD COMPONENT** (**FUNCTIONAL** and **HARDWARE**) has a **HISTORY** (UC 30) (see Figure 63). Usually it is linked to the history of the hardware board on which the component sits. There is a many-to-many relationship from **FUNCTIONAL BOARD COMPONENT** to **HISTORY COMPONENT** and also from **HARDWARE BOARD COMPONENT** to **HISTORY COMPONENT**.

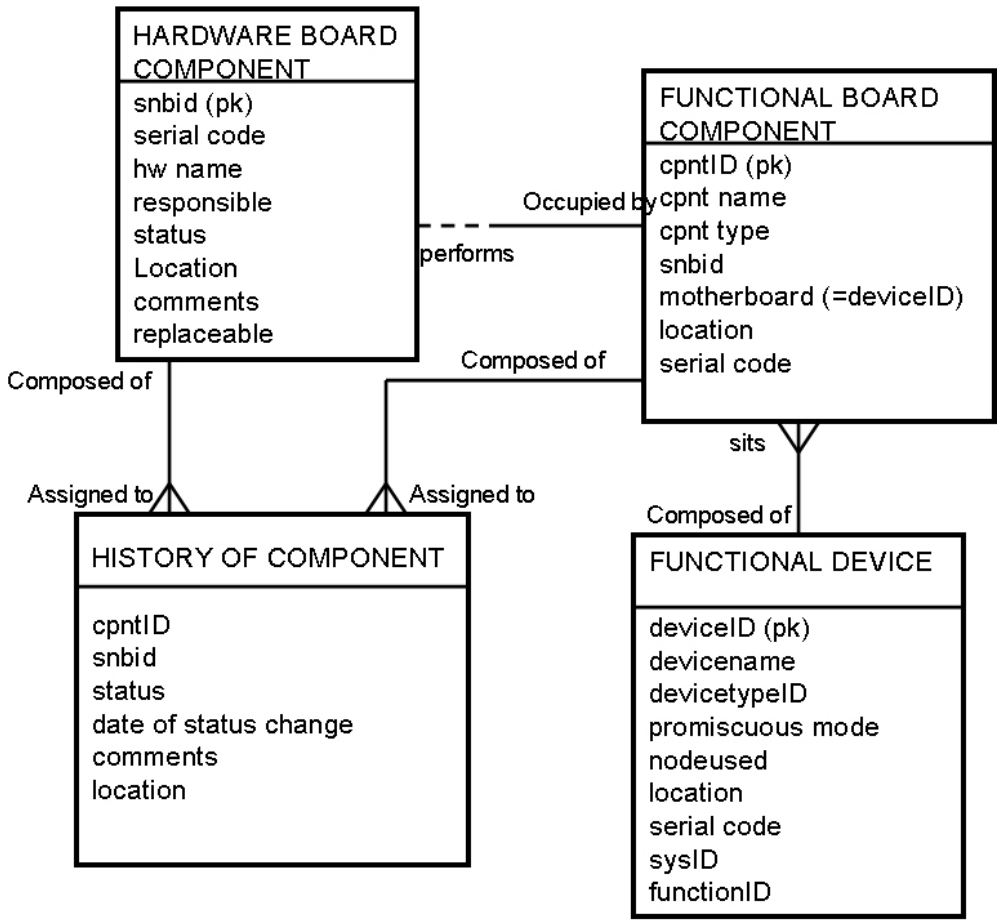


Figure 63. HISTORY COMPONENT representation.

5.5.2 Table schema

Figure 64 shows the table schema designed to represent history and inventory. Attributes (**created**, **user_update**, **author**, **terminal_name**) used for internal management have been added to the **FUNCTIONAL_DEVICE_TYPES and FUNCTIONAL_DEVICES** tables**.**

Also **FUNCTIONAL_DEVICE.nodeused** corresponds to the **enabled** attribute of the **FUNCTIONAL_DEVICE** entity.

- The **HARDWARE_DEVICES** table contains all the hardware devices. The **status** column represents the current status of the hardware device. The **serial code** is the primary key of this table. It identifies uniquely the hardware device.
- The **FUNCTIONAL_DEVICE_TYPES** table contains all the functional device types. The primary key (**devicetypeID**) is a sequence of number to avoid complex primary keys (see next section for explanations).
- The **FUNCTIONAL_DEVICES** table contains all the functional devices. The **serial code** column is a foreign key to **HARDWARE_DEVICES**.**serial_code**. The **devicetypeID** column refers to **FUNCTIONAL_DEVICE_TYPES**.**devicetypeID**. The primary key is **deviceID**, a sequence of numbers and not the **devicename (**which is a candidate key**)** for performance reasons. The comparison between numbers is

faster than the comparison between strings. The **nodeused** column is a flag indicating if the functional device is disabled (0) or enabled (1). The status is deduced from the status of the hardware device occupying the functional device. The subsystem column is described in section 5.6.4 It indicates which subsystem(s) a device is part of. It is used for navigability and partitioning reasons. The **node** column is explained in the next chapter.

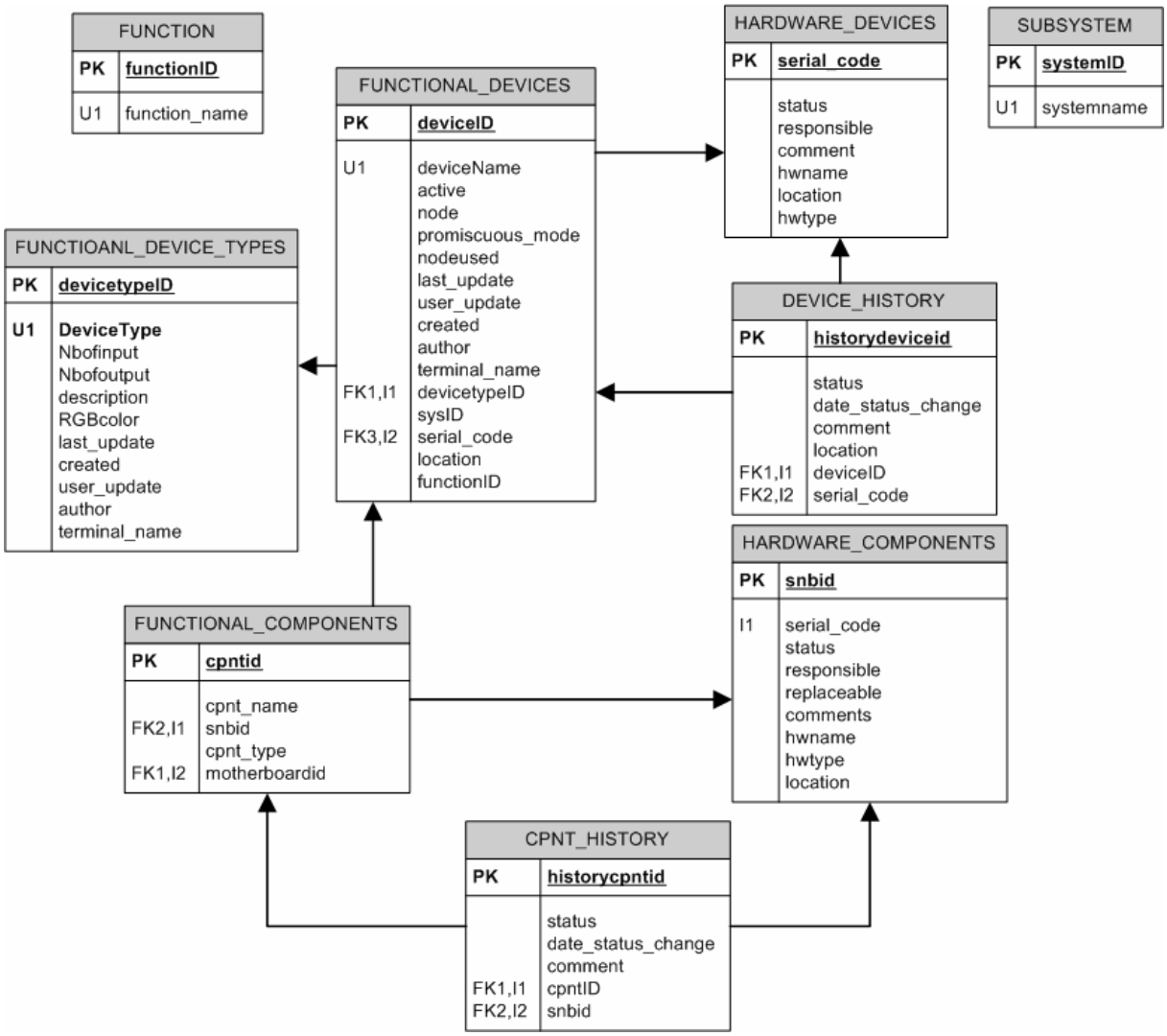


Figure 64. Table schema for the history and inventory data.

- The **DEVICE_HISTORY** table contains history of **FUNCTIONAL** and **HARDWARE_DEVICES**. The primary key is **historydevid**, a sequence of numbers to ensure uniqueness. **DeviceID** refers to **FUNCTIONAL_DEVICES.deviceID** and **serial_code** to **HARDWARE_DEVICES.serial_code**.

Similar table structures for components have been implemented. The main differences are:

- The **HARDWARE_COMPONENTS.snbid** column, a sequence of numbers, is the primary key instead of the **serial_code**. Not all the components have their own **serial_code**, however they are used in the connectivity.
- The **FUNCTIONAL_COMPONENTS.motherboardid** column refers to the

FUNCTIONAL_DEVICES.deviceID. It corresponds to the location of the component.

- **Nodeused** and **promiscuous_mode** have no meaning for a board component. There is no requirement to disable a board component.

Components and devices were separated for two reasons:

- The constraints are different, for instance the serial code is not defined for all the board components. A hardware device is replaceable but not necessarily a board component. It is easier to implement checking functions (check (or declarative) constraints, trigger) to keep consistency. First we tried to implement the constraint using check constraint feature[5]. If it is not possible, either it is implemented with a trigger or in the application code (as one single library will be used to interact with the connectivity and the history related tables in the CIC DB).
- Initially it was not foreseen to store board components. The need to also store board components appears late (end of 2005). If in the years to come, there is no need for this, the tables can be ignored[6]. The other part of the database is not affected.

5.6 Connectivity design

Storing the connectivity consists of describing the DAQ network and other subdetectors topologies. It has two levels, the macroscopic and microscopic connectivity.

5.6.1 Introduction

Connectivity can be viewed as the set of edges of a graph where a node is a pair (device, port). It uses functional devices. Let us assume there is a link between the functional devices A and B. A is occupied by the hardware A1882TYGG and B by GH6789JKJK. If B is replaced by the hardware GJKKKK789, there is still a link between A and B.

Relating the connectivity to hardware would have implied a change whenever there is a hardware replacement, which is not the case if the connectivity is considered as functional. We could also envisage considering hardware links, i.e. links between two hardware devices. However hardware links are stored in another database for traceability and security reasons as there are hardware devices exposed to the radiation area.

Connectivity consists of describing links between functional devices. A functional link corresponds to physical cables in the reality. However there is no mean to check that all the links have been inserted. It is up to the user to make sure that the all the links have been inserted properly.

5.6.2 Boot image: entity & relationship model

To generate the dhcp config file, the boot image of a PC in the farm or a TELL1 board must be known. The boot image is usually bound to the type of the device. However in some cases, it can be specific to a device. This information has been represented using two new entities, **DEVICE TYPE BOOTING** (UC 9) and **DEVICE BOOTING** (UC 9)**.** In both cases, the attributes are shown in Figure 65.

[5] Check or declarative constraint is specified when creating a table. For instance, check that the value of the column is always greater than 0 can be implemented using a check constraint.
[6] If tables are truncated, the application software will not be affected. It is like empty tables.

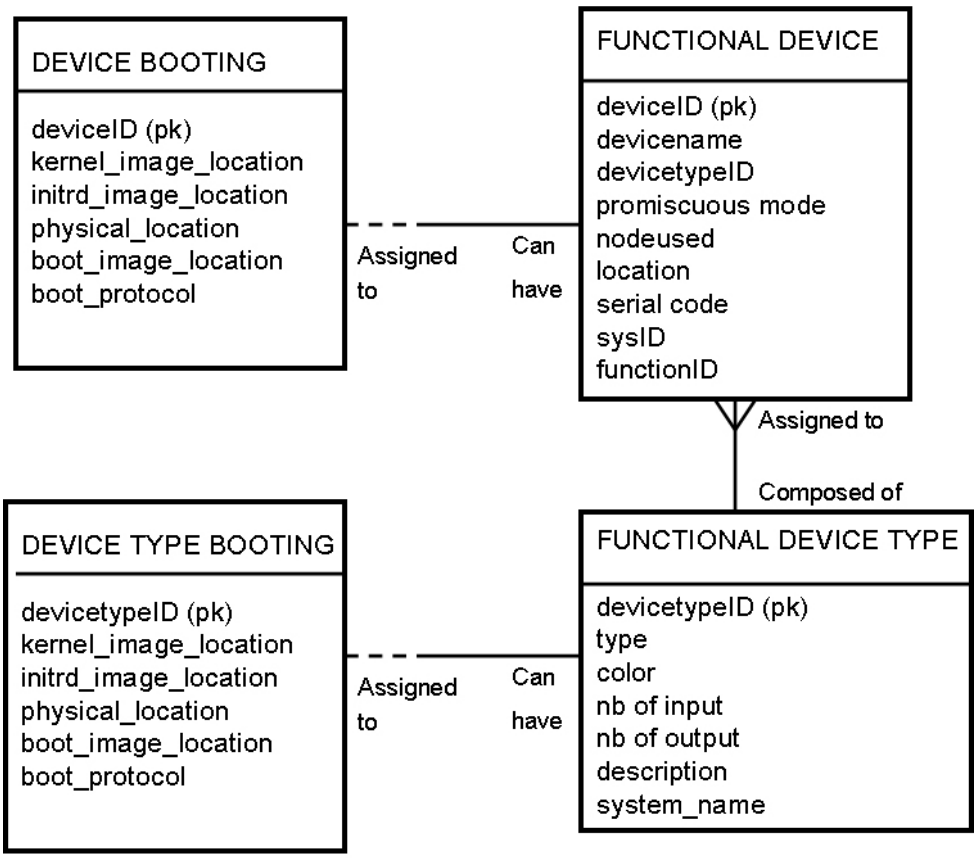


Figure 65. DEVICE BOOTING and DEVICE TYPE BOOTING models.

There is a one-to-one relationship from **DEVICE TYPE BOOTING** and **FUNCTIONAL DEVICE TYPE.** There is only one type of booting image per type, so **DEVICE TYPE BOOTING** is uniquely defined by the **devicetypeID**, which is also a foreign key to **FUNCTIONAL_DEVICE_TYPE.devicetypeID**. The same is true for **DEVICE BOOTING**.

5.6.3 Partitioning representation

At start up, the operator will select subdetectors to define a partition. The name of the sub-detectors is already defined as they are also used by the ECS for control and monitoring purposes. A subdetector can be run independently if only and only if it is connected to an output port of the TFC switch. A subdetector (used in the ECS) can regroup one or several parts of detector from a TFC system point of view. For our purposes, the partitioning process consists of associating a subsystem to the output port on a switch.

There are two methods to model partitioning.

1. This is the most intuitive one. It consists of defining the entity **OUTPUT_PORT** with these two attributes **output_port_nb** and **subsystem_name**.

Subsystem name	Output_port_nb
VELO	0
VELO A	1
VELO C	1
VELO	1
PUS	2
RICH	3
RICH	4
RICH1	3
IT	4
TT	5
OT	6
OT	7
OT A	6
OT C	7
RICH2	8
SPD/PS	9
ECAL	10
L0CALO	10
HCAL	11
MUON A	12
MUON C	13
L0MUON	12
L0DU	14
L0TRIGGER	10
L0TRIGGER	14
L0TRIGGER	12
L0TRIGGER	2

Table 9. Example of TFC switch output connectivity.

Table 9 shows the **OUTPUT_PORT** table following the port assignment described in document [3].

With this table, the correspondence between output port and subsystems is immediate.

2. The second method is based on the connectivity. Instead of storing the mapping between output port and subsystem name, we generate the destination table of the TFC switch. In other words, we find which devices are at the end of the TFC chain as described in Chapter 2, section 2.3. As a remark, the last devices are the TELL1 boards whatever the subdetector. Then using the **FUNCTIONAL_DEVICE.sysID** attribute, we know the subsystem(s) which are associated with the output port. The principle of the destination table and its creation will be explained in Chapter 6.

In this thesis, we have selected the second method because it is more elegant as it does not require creating another table to answer specific queries. Creating the **OUTPUT_PORT** table is redundant since this information can be derived from the **CONNECTIVTY** table. Also the second method follows the autonomic approach by reducing human intervention. The concept of destination table is useful in the DAQ system too.

5.6.4 Subsystem representation

5.6.5.1 Intuitive model

Getting all the paths that are part of a given subsystem which go through a given device is a requirement for navigability reasons.

A device can be part of one or of several sub-systems. The intuitive way to model is shown in Figure 66.

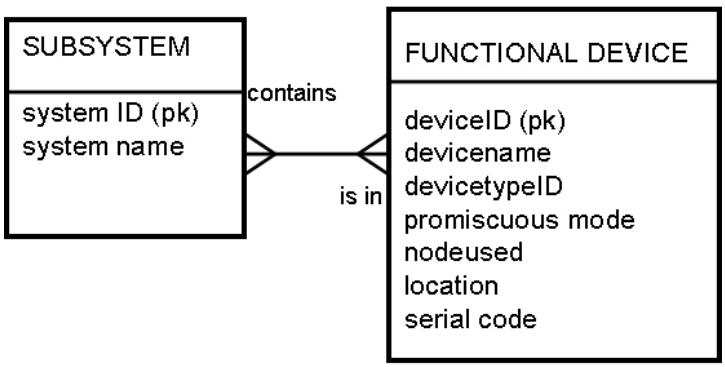


Figure 66. System table design.

Two new entities need to be modelled, **SUBSYSTEM**[7] and **FUNCTIONAL_DEVICE_ SUBSYSTEM. SUBSYSTEM** has a **name** and an **ID** (a number used to uniquely identify the sub-detetector). There is a many-to-many relationship between **SUBSYSTEM** and **FUNCTIONAL DEVICE**. A **SUBSYSTEM** may include **FUNCTIONAL DEVICE**s and a **FUNCTIONAL DEVICE** can be in several **SUBSYSTEM**s. So to model this association with tables, there will be an extra table called **FUNCTIONAL_DEVICE_ SUBSYSTEM** table**.**

The **FUNCTIONAL_DEVICE_ SUBSYSTEM** table will have many rows as a lot of devices will be included in different subsystems. This table will have two columns (**systemID** number, **deviceID** number), where **systemID** refers to **SUBSYSTEM.systemID** and **deviceID** refers to **FUNCTIONAL_DEVICE.deviceID**.

For instance, consider the following **FUNCTIONAL_DEVICE** MUON_CHAMBER_01 which belongs to the **SUBSYSTEM**s MUON_A and to L0MUON. The given device has two entries in **FUNCTIONAL_DEVICE_SUBSYSTEM.** One row will contain the **SUBSYSTEM.systemID** of MUON_A and the other one of L0MUON.

If the user wants to select all the devices which belong to MUON, the MUON_CHAMBER_01 will not be listed whereas it is part of MUON. It will not be part of the L0TRIGGER either. So two more rows need to be added to the **FUNCTIONAL_DEVICE_ SUBSYSTEM** table to declare MUON_CHAMBER_01 as part of {MUON, MUON_A, L0TRIGGER, L0MUON}.

All together four rows will be inserted in **FUNCTIONAL_DEVICE_ SUBSYSTEM** table**,** one for MUON A one for MUON, one for L0MUON and another one for L0TRIGGER**.**

[7] The SUBSYSTEM table will contain all the possible subsystems according to the ones defined in ECS. It prevents from attributing mistyped or unknown subsystems to devices. It is a security feature.

However performance is an issue. Programming the TFC switch should be done in less than a few seconds. Also getting paths between devices should be fast (less than 100 seconds for the Calorimeters as mentioned in Chapter 2).

5.6.4.2 Use of prime numbers

To improve the performance, the subsystem concept has been redesigned by exploiting the fact that the number of subsystems is quite low (less than 30). This number will be rather static as it is linked to the detector architecture and changing it would imply many changes in the design of the ECS. It is important to have an idea of the number of possible values that an entity can have as the prime numbers grow very fast.

In this design, we attribute particular numbers to subsystems according to the following algorithm.

Definition: A subsystem A is included in subsystem B if and only if all the devices of subsystem A are part of subsystem B.

To allocate an ID to a subsystem we proceed as follows:

1. Is the subsystem included in other subsystems?
2. If the answer is no, then we attribute it a prime number as an ID. A prime number can be attributed only once. If the answer is yes, then we attribute it a prime number and we multiply this value with the ID of the subsystem which contains it.

The principles of this algorithm are illustrated in Figure 67.

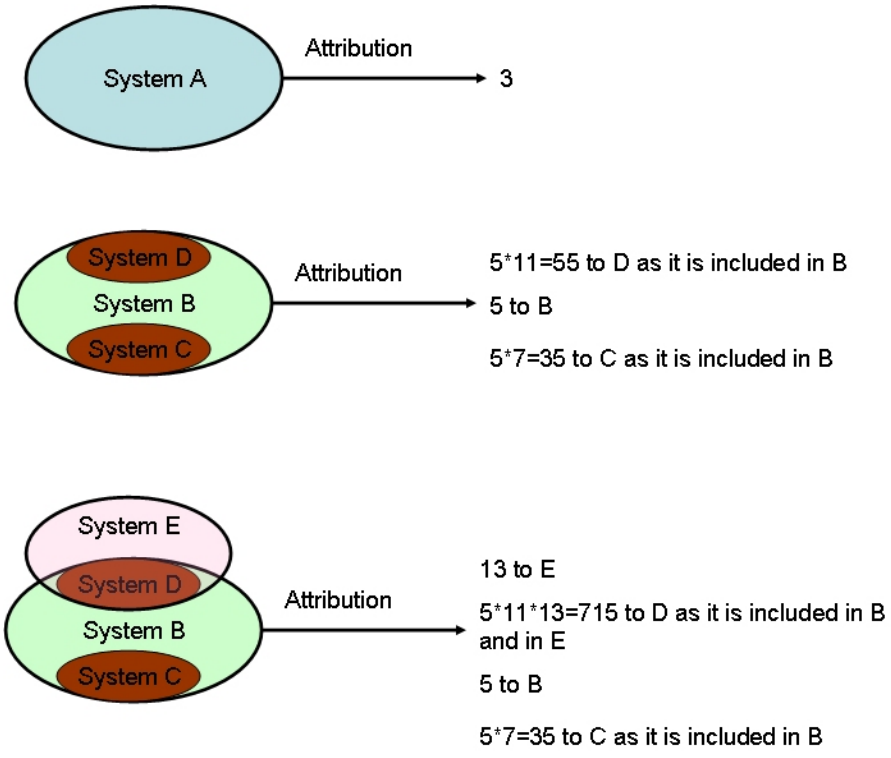


Figure 67. Attribution of prime numbers to subsystems. The last case is not used in the context of LHCb.

Referring to the third case shown in Figure 67, we have System D included in both System E and System B. We translate this information using the product of the two subsystemIDs. So when decomposing 715 (systemID of System D) in primes, we obtain 5, 11 and 13. We know that System D is part of System E (because of 13) and System B (because of 5).

However it is important to note that in the LHCb context, there is no such a case. In other words, a system has only one parent and consequently can be included at most in one subsystem. It is because the control system is hierarchical.

In the CIC DB, there is a table which contains one hundred prime numbers with their position as shown in Table 10. It can be extended by adding more prime numbers if needed. 1 is not considered as a prime number (all the numbers can be divided by 1).

Position	**Prime number**
1	2
2	3
3	5
4	7

Table 10. Extract of the prime number table.

To make the association, subsystem and devices, the user must specify only the smallest subsystem name (which cannot be divided any more). For instance, if a device is part of MUON A and part of L0MUON, only MUON A and L0MUON must be mentioned by the user. The other **SUBSYSTEM** (MUON and L0 TRIGGER) are automatically retrieved as the prime number decomposition of a number is unique.

The correspondence between subsystems and subsystem IDs can be predefined and stored in **SUBSYSTEM table** (see Table 11**)**.

Remark:

L0MUON is not included in the MUON system as they have different TELL1 boards so L0MUON is not included in MUON. (The same remark can be applied for the L0CALO).

Then using this **systemID** attribution, **FUNCTIONAL DEVICE.sysID** is filled by computing the product of the **SUBSYSTEM**s (which cannot be divided).

For instance, if the device is in both MUON A and L0MUON, the **sysID** is equal to 4757*6557=31191649.

In the other direction, getting all the devices which are part of a subsystem A associated to sysID_A is done by checking that mod (sysID, sysID_A) =0.

To update the **sysID** attribute, in the two cases (standard ERM model and primes model), the user needs to specify the new list of subsystems.

For instance, if a user has attributed VELO_A to a device identified by deviceid XX and normally this device is part of VELO_C, the update will be as follows:

- For the first method : delete all the rows in **FUNCTIONAL_DEVICE_ SUBSYSTEM** which have deviceid=XX and insert (XX,VELO_C); (XX,VELO);
- For the second method, update **FUNCTIONAL_DEVICE** set **sysID** =systemID(VELO_C) where **deviceid=XX;**

Subsystem Name	**Subsystem ID**	**Comments (in bold prime number which identifies a sub-sub-system)**
TFC	3	Subsystem which cannot be included
DAQ	5	Subsystem which cannot be included
VELO	7	Subsystem which cannot be included
VELO A	77	7***11** (included in the VELO)
VELO C	91	7***13** (included in VELO)
RICH	17	Subsystem which cannot be included
RICH 1	323	17***19** (included in RICH)
RICH 2	391	17***23** (included in RICH)
OT	29	Subsystem which cannot be included
OT A	899	29***31 (**included in OT)
OT C	1073	29***37 (**included in OT)
ST	41	Subsystem which cannot be included
IT	1763	41***43** (included in ST)
TT	1927	41***47** (included in ST)
ECAL	53	Subsystem which cannot be included
HCAL	59	Subsystem which cannot be included
PRS	61	Subsystem which cannot be included
MUON	67	Subsystem which cannot be included
MUON A	4757	67***71** (included in MUON)
MUON C	4891	67***73** (included in MUON)
L0MUON	6557	79***83** (included in L0TRIGGER)
L0CALO	7031	79***89** (included in L0TRIGGER)
PUS	7663	79***97** (included in L0TRIGGER)
L0TRIGGER	79	Subsystem which cannot be included
L0DU	7979	79***101** (included in L0TRIGGER)

Table 11. The Subsystem table with its subsystemIDs.

5.6.4.3 Performance comparisons

The tests have been made on a central Oracle database 10g installed by CERN Central Database Services. The workstation which hosts the database, is a SUN 280 R. The operating system is SOLARIS/RARP. The database service is a public service, shared between users from many experiments, not only LHCb.

I have executed the following SQL queries from two Oracle SQL*plus (i.e. two sessions). Autotrace (an Oracle tool to analyze queries) was set to TRACEONLY mode.

- Select t.**devicename** from **FUNCTIONAL_DEVICES** t, **SUBSYSTEM** l, **FUNCTIONAL_DEVICE_SUBSYSTEM** s where t.**deviceid**=s.**deviceid** and s.**systemid**=l.**systemid** and l.**system_name**='XXX' for the first method;
- Select t.**devicename** from **FUNCTIONAL_DEVICES** t, **SUBSYSTEM** l where mod (t.**sysid**,l.**systemid**)=0 and l.**system_name**='XXX' for the second method;

In both cases, 'XXX' corresponds to a subsystem name.

I have progressively increased the total number of devices. The Table 12 presents the results of the tests.

Number of devices[8]	**System name**	**Number of rows returned**	**Execution time Method 1 (avg, sec) Standard ERM model**	**Execution time Method 2 (avg, sec) Primes model**
7790	DAQ	2962	0.04	0.02
15578	MUON_C TFC	14 6676	0.09 0.06	0.01 0.04
124610	OT MUON	2160 224	1.09 0.08	0.08 0.04
249218	VELO DAQ	2944 94784	4.04 8.01	2.09 7.02
498434	MUON L0TRIGGER L0TRIGGER RICH VELO DAQ	896 640 1536 5184 5888 189568	8.01 10.07 10.00 4.08 7.00 24.04	14.02 11.00 8.03 3.02 5.06 21.08
996864	DAQ ST VELO	379136 12288 11776	40.01 18.56 22.06	36.07 22.50 28.00

Table 12. Comparison between the two methods.

From the results, one notices that Method 2 is more efficient as long as the number of devices is not higher than 500,000. It is due to the fact that Oracle indexes cannot be used if a function is used on a column. So the second method always performs a full scan of the FUNCTIONAL_DEVICE table because of mod (…) unlike the first method.

In my case, this query will be performed against the destination table of the TFC which has less than 60,000 rows as it will be shown in the Chapter 8. And in that case, the method 2 is faster.

5.6.4.4 Limitations of the prime number algorithm

The problem

One can notice that the **systemID** grows very quickly as it is a prime number. So if there are too many subsystems, the model cannot be used. The aim of this section is to determine an upper limit on the number of subsystems, above which the prime number algorithm breaks down.

The inclusion of subsystems is determined by the FSM hierarchy. In the FSM hierarchy, for instance, the VELO consists of two subsystems VELO_A and VELO_C.

The number of the TFC output ports will fix the number of leaves of the tree (in other words the level of granularity). For instance, all the devices of VELO_A are driven by the output 1 of the TFC switch (see Table 9). If in the FSM hierarchy VELO_A is split into 3 parts for instance VELO_A_1, VELO_A_2 and VELO_A_3, these 3 subsystems are not needed. Indeed from a partition point of view, selecting VELO_A_1 or VELO_A is the same, the TFC

[8] The fact that the figures are precise has no particular meaning.

switch will program in the same way. So the subsystems VELO_A_1, VELO_A_2 and VELO_A_3 will be not stored in the SUBSYSTEM table.

So the deeper the tree is (or levels of inclusion), the bigger the systemID will be. In this part, we try to find the boundaries of this model, in terms of the depth of the tree design and the number of output ports. So there are two essential variables, the number of outputs of the switch which determines the number of leaves and the depth of the tree.

The **systemID** is an Oracle number, whose precision is less than 10^{38} digits [4].

Convention: Let us note P(n), the function which gives the n^{th} prime number, where n is an integer>1. (so we have P(1)=2, P(2)=3, P(3)=5, etc.).

We present three scenarios:

1. In the first one, there is no inclusion. It is the best case scenario. None of the systems has a parent. So each system is identified by a prime number. This scenario shows the maximum values that one of the two entities can have, in the case the method with the prime numbers should replace a N:M relationship. In our case, it will also correspond to the maximum number of output ports of the TFC switch.
2. In the second one, there is one level of inclusion only. A parent has only two children. It is the current design in LHCb as an output port of the TFC switch serves a subsystem which is included at most in one subsystem. Then we evaluate the maximum number of output ports, which also corresponds to the maximum of child subsystems (with one parent) which can be defined.
3. In the third one, we simulate a worst case scenario, by maximizing the depth of the tree (i.e. the levels of inclusion). At each level of the tree, a subsystem is included. The advantage of this tree is a relation can be derived between the number of leaves and the depth of a tree. We assess the maximum number of subsystems which can be handled.

First scenario the FSM subsystem tree corresponds to the TFC partitions

In that case, there is no subsystem inclusion. The subsystems displayed to the user correspond exactly to the TFC partitions. In this case, the maximum number of subsystems corresponds to the number of prime numbers less than 10^{38}, roughly $2.30*10^{34}$ prime numbers, which is the upper limit on the number of subsystems [5].

However there is a limit fixed by the number of output ports of the TFC switch. Nowadays, on the market the biggest switch has 1200 ports. It is the Force Ten switch used in the DAQ system. So if there is no subsystem inclusion, the model works fine with the biggest switch as $2.30*10^{34}$ >1200. It also implies that this model can be used to replace a N:M relationship model if one of the two entities has less than $2.30*10^{34}$ values.

Second scenario each subsystem is grouped by two

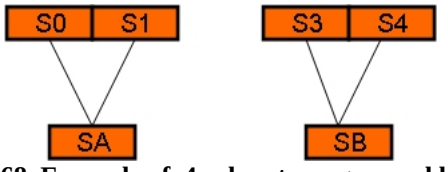


Figure 68. Example of 4 subsystems grouped by two.

Let us assume that the TFC switch has N outputs and subsystems are grouped per two (Figure 68).

If N is odd, then there is one inclusion of 3 subsystems. There is only one level of inclusion as it is the case in the current implementation. Let us found the maximal value of N.

There are N children and E[N/2] parents. The biggest systemID will be P(E[N/2]*P(N+E[N/2]). If N is even, N can be written as N=2k and the previous formula is P(k)*P(3k). If N is odd, N can be written as N=2k+1 and the previous formula is the same as for even numbers, i.e., P(k)*P(3k). P(k) is equivalent to klog(k) and P(3k) to 3kln(3k), so P(k)*P(3k) is equivalent to $k^2\ln(k)*\ln(3k)$ [4].

And $k^2\ln(k)*\ln(3k)$ must be have at most 39 digits . If we note $f(p)= p^2\ln(p)*\ln(3p)$, f is an increasing function as product of increasing functions.

We have $f(10^{17})=10^{34}*39.14*40.24=1.575093*10^{37}$, $f(5.10^{17})=4.26*10^{38}$.(39digits), $f(8.10^{17})=1.11*10^{39}$ (40 digits). So $k=8*10^{17}$ is a good approximation. And $N=16.10^{17}$.

There are some years to go by before getting a switch with 16.10^{17}output ports!

Third scenario Example of a tree where one leaf is added at each level

The maximum depth of this tree design as shown in Figure 69 can be computed given the number of leaves which is equal to the number of output ports.

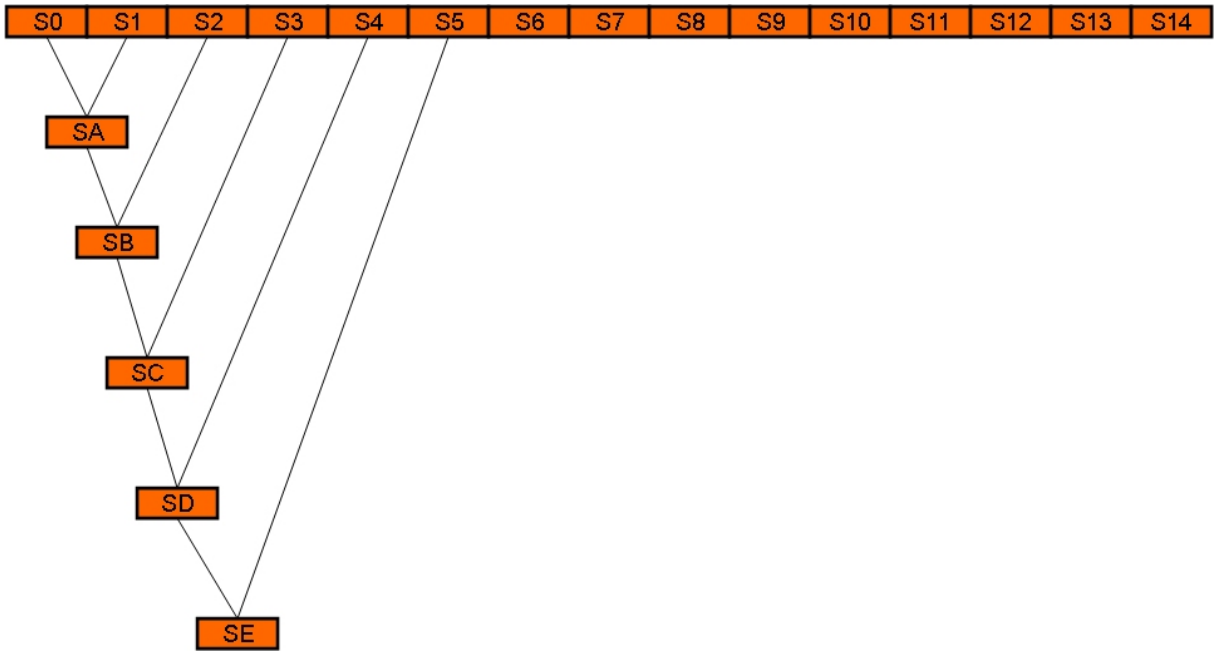


Figure 69. Example of a representation of a tree of depth equal to 5 and with 15 leaves.

The total number of nodes in this type of tree design is equal to 2N-1 where N corresponds to the number of leaves ($N\geq 2$) and the depth is equal to N.

Proof of the formula by induction on N

N=2

Figure 70 shows how the tree looks like. There is in total 3 nodes (2 leaves + one parent node). And 2N-1=2*2-1=3. The depth is 1 (2-1). So the formula is true for N=2.

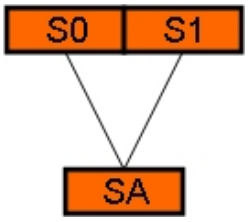


Figure 70. A tree of depth 1 with 2 leaves.

Let us assume that it is true for N, let us show it for N+1.

N->N+1

Figure 71 shows how to build a tree of N+1 leaves from a tree of N leaves. Two nodes have been added (represented in green). SN+1 represented the (N+1) th leaf of the tree. Then, this node is joined to the tree of N leaves by adding the node SP+1. This node has two children, SN+1 and SP, the top node of the tree with N leaves.

By assumption, the tree with N leaves has 2N-1 nodes. To build the tree with N+1 leaves, we have added two nodes (SN+1, SP+1). So the tree with N+1 leaves has (2N-1) + 2 = 2N+1 = 2(N+1)-1. So the formula is true for N+1.

The depth of the tree with N leaves is N-1. The tree with N+1 leaves has an extra level with the SP+1 node. So the depth of the tree with N+1 leaves is (N-1) +1=N. So the formula is true for N+1.

Thus it is true for any N (q.e.d).

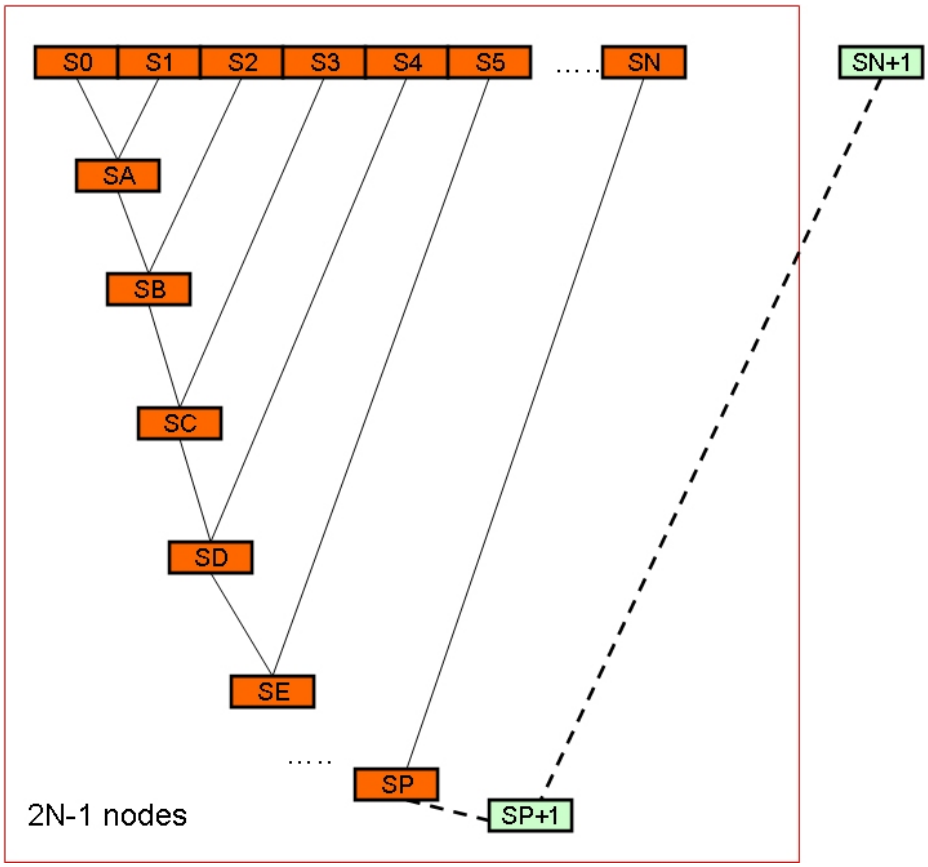


Figure 71. Building a tree of N+1 leaves from a tree with N leaves.

The next step is to quantify the subsystemID number limit according to N.

Case where N=16 (the number of TFC output switch)

The **sysID** is an Oracle number, which cannot be greater than 10^{38}.

Let us assume that the TFC switch has 16 output ports, so N=16. There are 2*16-1=31 systemIDs and the depth is equal to (16-1) =15 levels.

The root of the tree will get the smallest ID, which is the smallest prime number. It is then equal to 2.

The two children of the root will get 2*3=P (1)*P (2) and 2*5=P(1)*P(3) as systemIDs. At the second level of inclusion, P(1)*P(3)*P(4) and P(1)*P(3)*P(5) will be used. At each level, two new prime numbers are needed.

So at the last level, (level=N-1) of the tree the two subsystem IDs will be $P(k)\prod P(2n+1)$ where n is an integer between 0 and 14 (15-1), as there are 15 levels and k is either 30 or 31 as there are two subsystems which are included iteratively in 15 subsystems. In other words it is the product of over P(n) where n is an odd number between 1 and 31.

So the biggest systemID will be $P(31)*\prod P(2n+1)$ where n varies from 0 to 14. We have P(31)=127, P(29)=109,…, P(1)=2. So the biggest systemID is 42554922123902186586179 35, which can be stored in the CIC DB as it has 25 digits.

Computing the maximal value of N with this type of tree

Let us assume that the TFC switch has N outputs. The biggest systemID will be equal to $P(2N-1)*\prod P(2n+1)$ where n is an integer between 0 and (N-1)-1=N-2.

After computations using the list of the 1000 first primes [5], I found that the maximum value of N is equal to **21**. For N=22, $P(2N-1)*\prod P(2n+1) = 1{,}55736049*10^{38}$, and P(2N-1)=P(43)=191. This number has 39 digits.

To conclude, to go for this representation, it is preferable that the number of level of inclusions is not too high, strictly less than 22.

5.6.4.5 Link type representation

The **LINK TYPE** (UC 8) has been modeled with a similar concept. A simple link type such as the TFC signal will be associated with a prime number (**link_nbr** attribute). A composite link type (TFC signal and data signal), which is a set of simple link types, will be associated with the product of the prime numbers (**link_nbr**) of the links in the set. There is at most one level of inclusion. So it allows having 16.10^{17} different types of links referring to the second scenario in the previous section. Here again, the number of link types is rather low, less than 20 in total. It is also a static number over the years. So the method with the prime numbers can be applied.

5.6.4.6 Function representation

The **FUNCTIONAL_DEVICES**.**functionID** has been modelled using prime numbers. There is a table FUNCTION (see

Figure 64) which contains the possible functions. There are a limited numbers (less than 20) and mainly used for the DAQ (DNS, NFS, TFTP, DHCP have currently defined). Each function is attributed a prime number as an ID. If there is no function the attribute **functionID** is 0. The **FUNCTIONAL_DEVICES**.**functionID** corresponds to the product of the functioned, it fulfils. For instance, if a controls PC hosts both a DHCP server and a DNS server and the **functionID** of the DNS is 2 and the **functionID** of the DHCP is 3, then 2*3=6 corresponds to the **functionID** of this controls PC.

Here there is no level of inclusion, so it corresponds to the first scenario described in 5.6.4.4. It means that $2.30*10^{34}$ different functions can be stored, which is by far more than the possible functions we had (less 20 for the moment).

5.6.5 Entity & relationship

A **LINK** (UC 8) is a connection between two ports of two functional devices. To program the routing tables and the TFC switch and also to get paths between 2 devices, the port should be

specified. A link carries data, a tfc signal, a data signal, a high voltage signal, or it can be a mixture of tfc_signal and data signal (UC 8).
Three entities have been modelled, a port, a link type and a link.

A port is a generic concept which covers connectors and network interfaces. A port is where you can plug a cable to it. It has attributes bound to the hardware device and others to the functional device.

A **HARDWARE PORT** (see Figure 72) belongs to a **HARDWARE DEVICE** (UC 12). There is a many-to-one relationship from **HARDWARE PORT** to **HARDWARE DEVICE**. It is also uniquely identified by a **port number** (UC 8), a **type** (UC 12) and a **dataflow way** (input or output) (UC 8) and a **serial code** which corresponds to the **HARDWARE DEVICE** it belongs to. In the case of an Ethernet port (so there is no concept of input or output), the dataflow way is defined according to the data coming from the detector. It can have a **burnt internal address** (corresponds to the real MAC address) (UC 12) and a **MAC address** (UC 12), such as the port of a DAQ switch. These attributes depend on the hardware device.

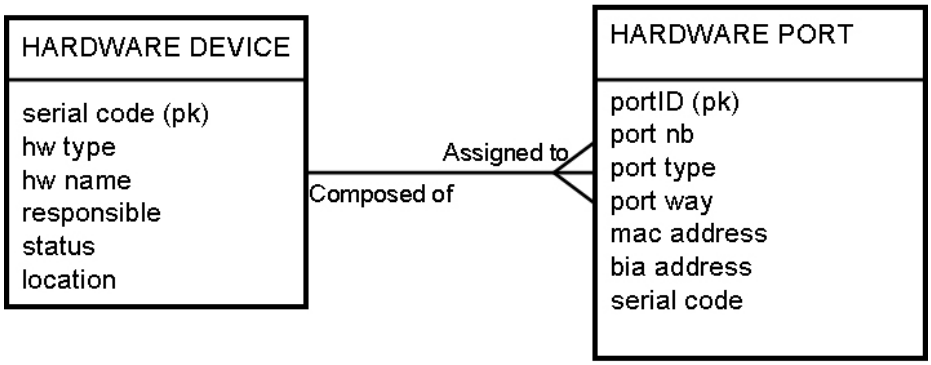


Figure 72. HARDWARE PORT model.

A **FUNCTIONAL PORT (**see Figure 73**)** (UC 8) belongs to a **FUNCTIONAL DEVICE. There is a many-to-one** relationship model from **FUNCTIONAL PORT** to **FUNCTIONAL DEVICE**. It is also uniquely identified by a **number** (UC 8), a **type** (UC 12) and a **dataflow way** (same concept as in hardware) (UC 8) and a **deviceID.** It has an **administrative status** (UC 12). It can be up or down. Some attributes are specific to the DAQ system such an **IP address** (UC 12), a **phy** (type of cable which can be plugged) (UC 8) and a **speed** (UC 8). In some cases, a port can be viewed as two logical interfaces. One interface is used for data acquisition and another one for the control. A **pxi booting** (PCI EXtensions for Instrumentation) (UC 11) flag specifies which logical interface will be used for booting. All these parameters will not change after a replacement of hardware.

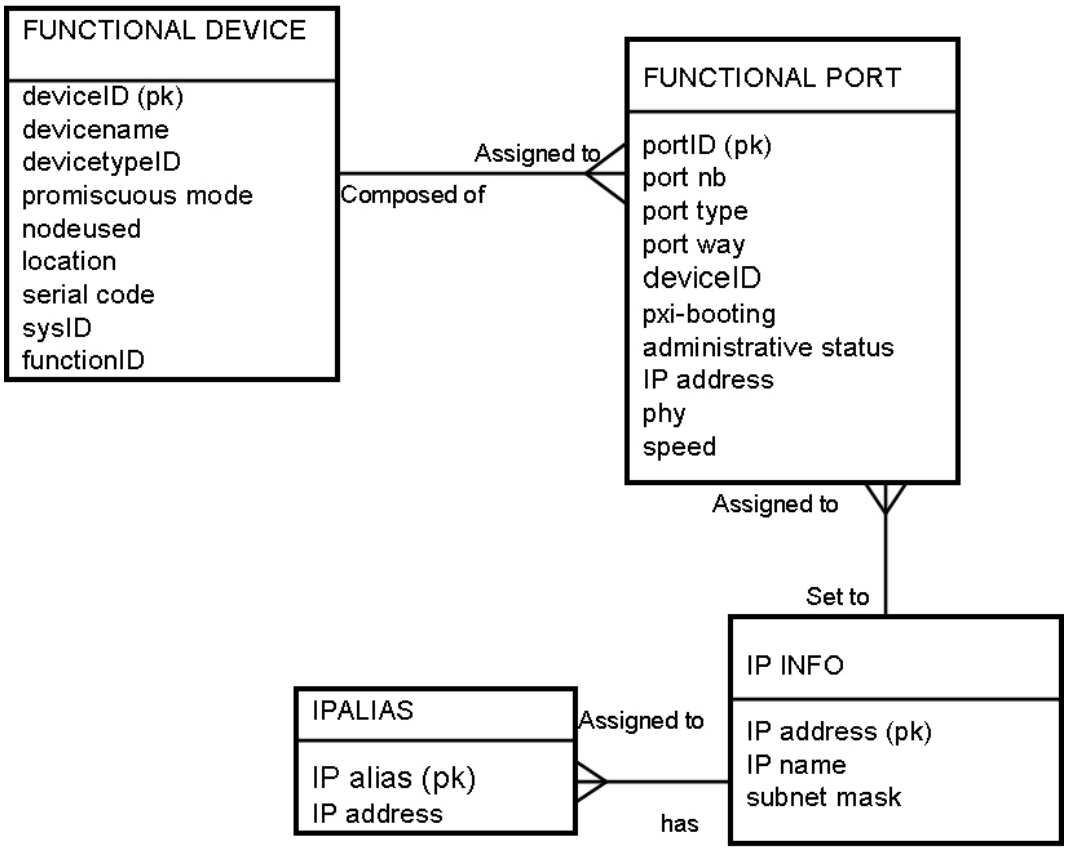


Figure 73. FUNCTIONAL PORT model.

An **IP INFO** (UC 12) as shown in Figure 73 can be considered as an entity too. It is part of a subnet so it has a **subnet mask** (UC 9). Also an **IP INFO** is associated with a **name** (UC 9). There is a one-to-one relationship between **IP INFO** and **name**. The same IP address can be attributed to several functional ports.

An **IPALIAS** (UC 14) as shown Figure 73 is an entity. An IPALIAS is assigned to an IP address. One IP address can have several IP aliases (UC 9).

A **LINK TYPE** (see Figure 74**)** (UC 8) has a **name**. It can be a simple link type, i.e. a TFC signal or it can be a composite link type, i.e. TFC signal and data signal. The composition of a link is associated with **link_nbr (**see section 5.4.4.5 for explanations**).**

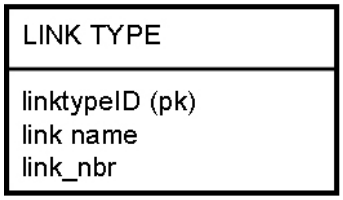


Figure 74. LINK TYPE model.

The **CONNECTIVITY** (UC 7) entity (see Figure 75**)** is between two **FUNCTIONAL PORTS**. A link can be **uni**- or **bi-directional** (UC 8). A link can be **enabled** or **disabled** (UC 13) (by performing some masking operations in electronics registers). The

CONNECTIVITY entity has a **LINK TYPE** (UC 8). There is a many-to-one relationship from **CONNECTIVITY** to **LINK TYPE**.

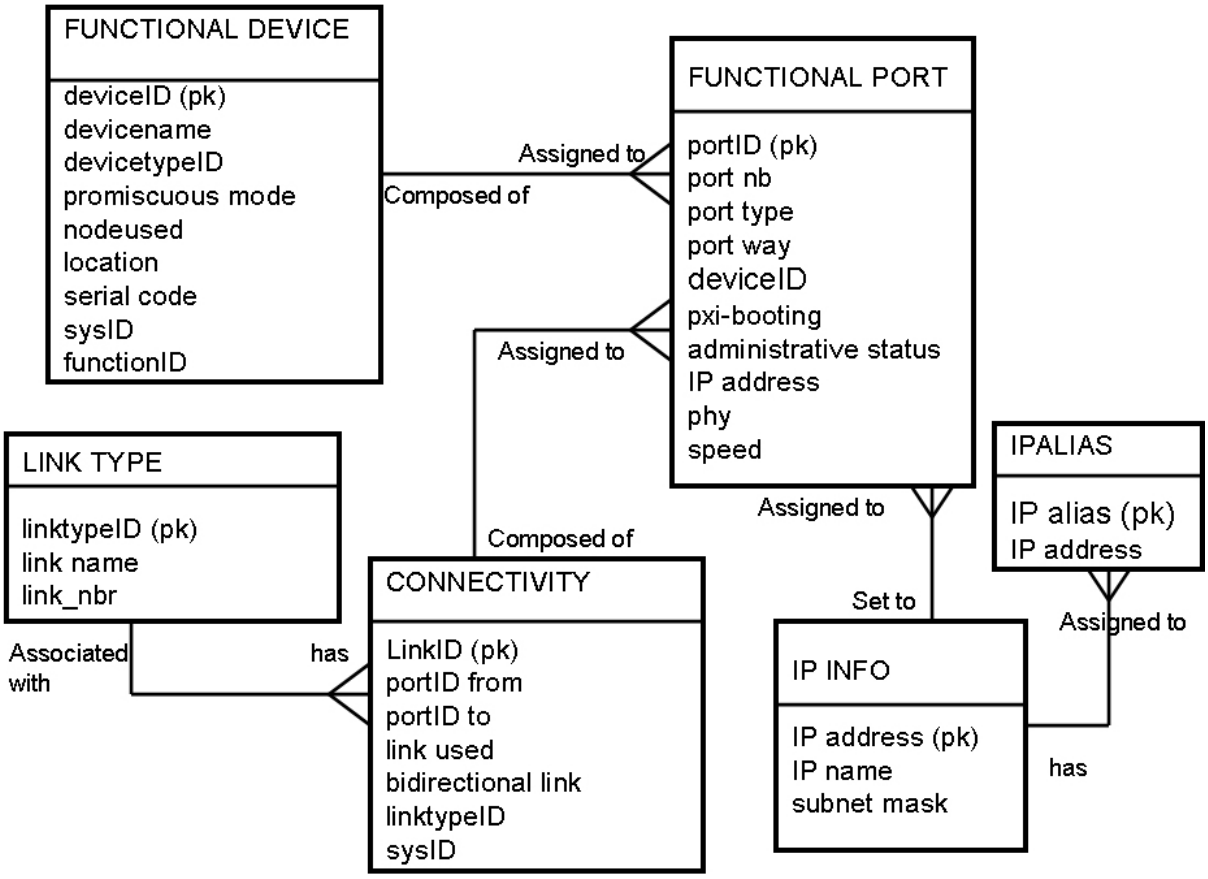


Figure 75. LINK model.

5.6.6 Board components

In some cases, the connectivity of a board must be stored (UC 28). It is the microscopic level. It has been modeled in a similar way as the connectivity between functional devices but it is less complicated. Indeed for board components, the port entity has been removed as there are no details required about the "port" of a chip. It is just necessary to know which port(s) of a functional device, the chip is connected to. The different link types of a microscopic link are the same as the ones defined for macroscopic links (otherwise tracking paths is not possible). The concept of subsystem is removed.

5.6.7 Table schema

Figure 76 shows the table schema of the connectivity part.

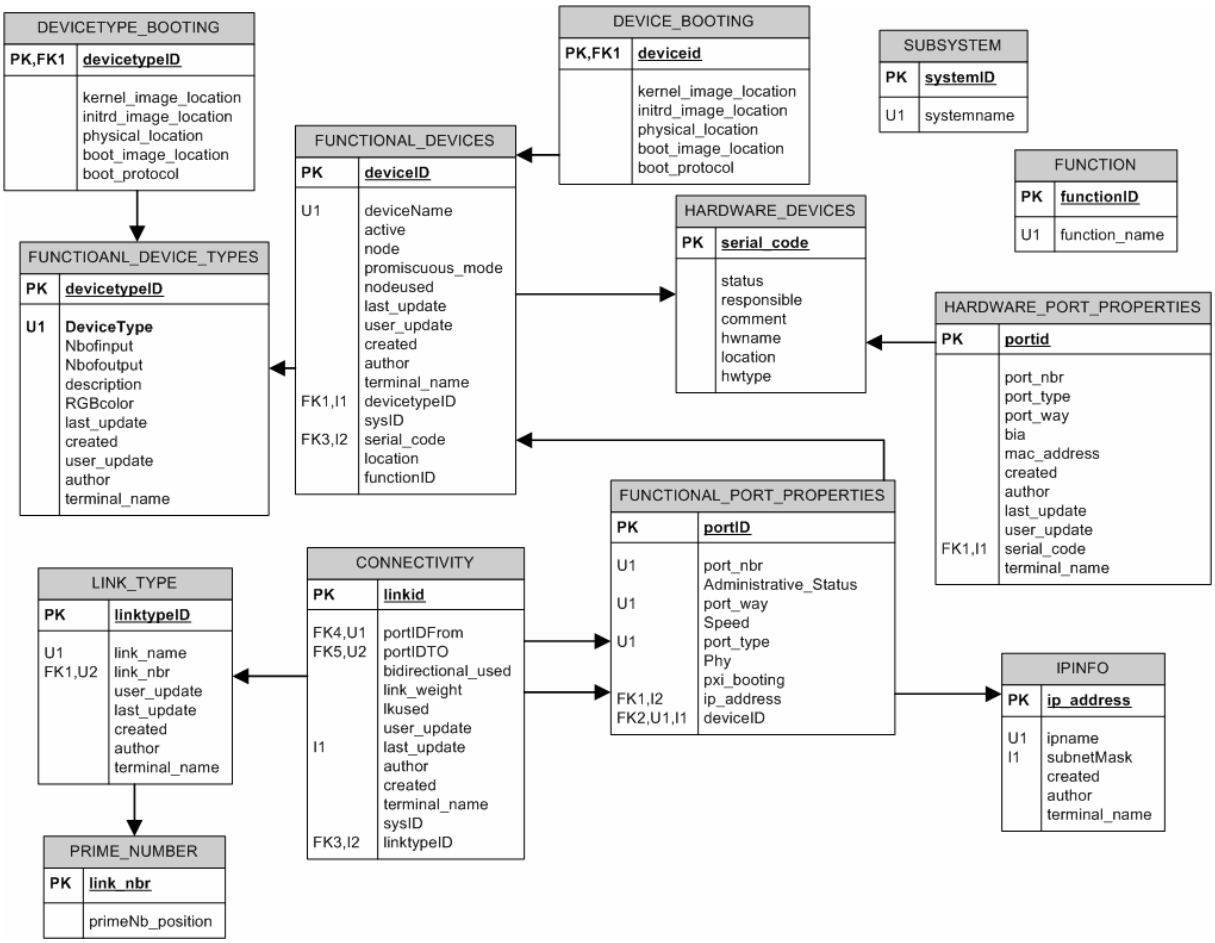


Figure 76. Connectivity table schema.

The **SYSTEM_NAME_LIST** table contains the list of the subsystems.

The **DEVICETYPE_BOOTING** table (resp. **DEVICE_BOOTING**) contains boot image information for a certain device type (resp. device). The **devicetypeid** (resp. **deviceid**) is a foreign key to **FUNCTIONAL_DEVICE_TYPES**.**devicetypeid** (resp. **FUNCTIONAL_DEVICES**.**deviceid**).

The **portid** column (sequence of number) is the primary key of the **FUNCTIONAL_PORT_PROPERTIES** table. It is to avoid having a complex primary key composed of (**deviceid**, **port_nbr**, **port_type** and **port_way**). The **deviceid** column is a foreign key on **FUNCTIONAL_DEVICES.deviceid** so that we know to which functional device this port belongs.

The **ip_address** column is a foreign key to **IPINFO**.**ip_address**.

Apart from the **ip_address**, the same remarks can be made for the **HARDWARE_PORT_PROPERTIES** table.

The **linktypeid,** a sequence of numbers is the primary key of the **LINK_TYPES** table to allow an easy update of the link type. If we need to remove a simple link from a composite link, we just update the **LINK_TYPE**.**link_nbr** column table. It does not affect the connectivity table.

The **CONNECTIVITY** table contains all the links between device ports. **PortIDFrom** (resp. **PortIDTO**) column specifies the start point (resp. end point) of the link. Both of them are foreign keys to **FUNCTIONAL_PORT_PROPERTIES**.**portID**.

The **bidirectional_link** column is a flag, 0 if the link is unidirectional. The **lkused** column is also a flag, 1 if it is used. The **linktypeID** indicates the type of the link. It is a foreign key to

LINK_TYPE.**linktypeID.** The **link_weight** column is explained in the chapter. It is used to get the paths too.

The **sysID** column specifies to which subsystem(s) this link belongs. Normally, one can guess to which subsystem(s) a link belongs. Indeed assume that there is a link is between device A, port 1 and device B, port 2. Device A is part of {subsystem_1, subsystem_2, subsystem_3} and device B is part of {subsystem_1, subsystem_2}. So the link is part of the intersection of the two ensembles, i.e. {subsystem_1, subsystem_2}. For performance reasons, the **sysID** has been added to the connectivity table. It avoids doing this computation as it is often queried. However it is important to note that to avoid any inconsistency, this column is accessed only in read-only mode, i.e. the user or the application program cannot write into this column.

Figure 77 presents the table schema for the board connectivity.

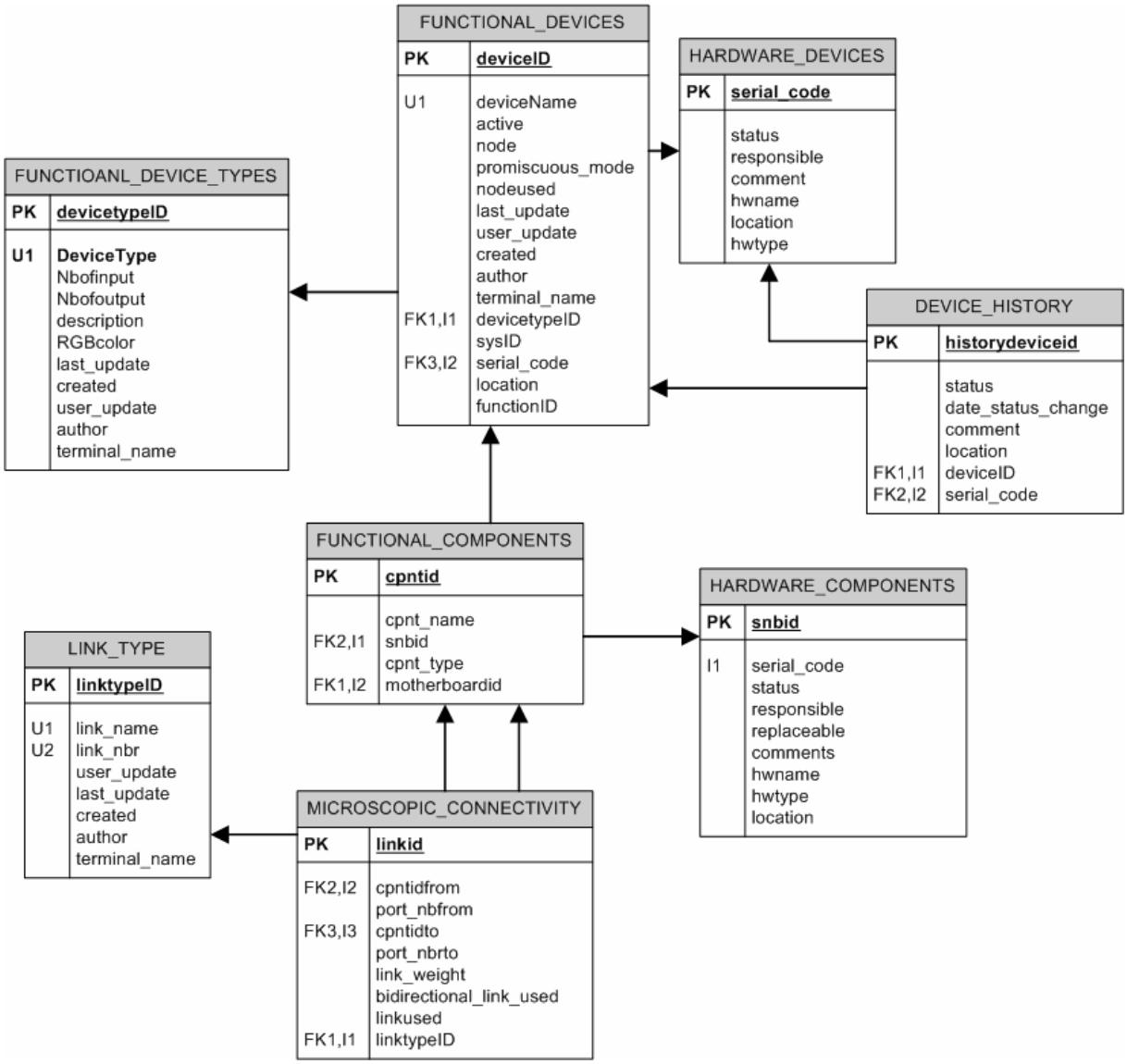


Figure 77. Board connectivity table schema.

5.6.8 A more complex table schema

LHCb is a complex environment. The table schema produced looks simple. But to reach this level of simplicity was not obvious. Also one could have created a **FUNCTIONAL_DEVICE_TYPE** table for each device type table so that it is possible to define a precise structure of device types. But the main inconvenience of this schema is whenever there is a new device type; one has to define another table. So the number of tables can grow quite quickly.

Another alternative for the **DEVICE_HISTORY** table would have been to design a **HARDWARE HISTORY** entity and a **FUNCTIONAL HISTORY** entity**.** But it would have been a problem when a hardware device is IN_USE.

If there is one history table, the content would have been like as shown in Table 13. 12456 is the **deviceID** of MUON_TELL1_12.

deviceID	Serial code	Status	Date of change	location
12456	XDG6FDG77	IN_USE	2006/07/21	DU78RC89SL9
NULL	GHOFD89878	SPARE	2006/05/18	Build2/Room5
12456	NULL	NONE	2008/10/23	DU78RC89SL9
NULL	XDG6FDG77	IN_REPAIR	2008/10/23	Frascati
12456	GHOFD89878	IN_USE	2008/10/24	DU78RC89SL9

Table 13. Content of the history table.

If there are two history tables, the content would have be as presented in Table 14.

Serial code	Status	Date of change	location
XDG6FDG77	IN_USE	2006/07/21	DU78RC89SL9
GHOFD89878	SPARE	2006/05/18	Build2/Room5
XDG6FDG77	IN_REPAIR	2008/10/23	Frascati
GHOFD89878	IN_USE	2008/10/24	DU78RC89SL9

Deviceid	Status	Date of change	location
12456	IN_USE	2006/07/21	DU78RC89SL9
12456	NONE	2008/10/23	DU78RC89SL9
12456	IN_USE	2008/10/24	DU78RC89SL9

Table 14. Content of the two history tables: on the top the HARDWARE HISTORY table and on the bottom, the FUNCTIONAL HISTORY table.

So the main disadvantages of the having **HARDWARE HISTORY** and **FUNCTIONAL HISTORY** are:

- There are more rows created. Whenever a hardware device is IN_USE, there is one row inserted in the **HARDWARE HISTORY** table and another one in **FUNCTIONAL HISTORY** table.
- There is a need of a reference between the two tables as we do not know which functional device is occupied by which hardware device, especially when two functional devices got down the same day. The location can be the same for two functional devices. So if the reference is added, it turns out to be model 1.
- The queries to get the history of a functional device or of a hardware device are more complex than in the first model.
  First model: select * from DEVICE_HISTORY where deviceid=:devid (or serialcode=:hwcode)

Second model: select * from HW_HISTORY t, FCTAL_HISTORY e, where e.deviceid=:devid and e.date of change=t.date of change and e.location=t.location
(However this is not sufficient!)

Another temptation was to let the users or applications programs store paths. But it would have been too rigid and difficult to maintain. Moreover there would be too many paths. It is more elegant to generate paths dynamically from the **CONNECTIVITY** table. In that sense, it follows the autonomics principles.

5.7 Verification of the completeness of the table schema

Let us take all the use cases defined in Chapter 4. Table 15 shows which tables and columns are used to get the information to fulfill the use cases. When no column is specified, it means that all the columns are needed. All the tables shown in the different table schemas are used and all the use cases have been satisfied at the level of the database schema. In some cases, use cases needs extra tables which are internal and dynamically filled. They are hidden to the users. These tables are used to get paths between devices and will be explained in detail in the next chapter.

Information type	**Use case number**	**Answer in Tables (columns)**
RECIPE	UC 1	RECIPES RECIPE_TAGS ITEM RECIPE_DATA HIERARCHY REFERENCES
	UC 2	RECIPES RECIPE_TAGS ITEM RECIPE_DATA HIERARCHY REFERENCES
	UC 3	RECIPES RECIPE_TAGS ITEM RECIPE_DATA HIERARCHY REFERENCES
	UC 4	RECIPES RECIPE_TAGS ITEM RECIPE_DATA HIERARCHY
	UC 5	RECIPES RECIPE_TAGS
	UC 6	FUNCTIONAL_DEVICES FUNCTIONAL_PORT_PROPERTIES CONNECTIVITY

	UC 7	CONNECTIVITY (lkinfo)
NETWORKING	UC 8	Other internal tables (see next chapter) FUNCTIONAL_PORT_PROPERTIES IPINFO HARDWARE_PORT_PROPERTIES FUNCTIONAL_DEVICES
	UC 9	FUNCTIONAL_DEVICES FUNCTIONAL_PORT_PROPERTIES IPINFO HARDWARE_PORT_PROPERTIES IPALIAS DEVICETYPE_BOOTING Other internal tables (see next chapter)
	UC 10	FUNCTIONAL_DEVICES DEVICE_HISTORY CONNECTIVITY Other internal tables (see next chapter)
	UC 11	FUNCTIONAL_DEVICES HARDWARE_DEVICES FUNCTIONAL_PORT_PROPERTIES IPINFO HARDWARE_PORT_PROPERTIES DEVICE_HISTORY CONNECTIVITY Other internal tables (see next chapter)
	UC 12	FUNCTIONAL_DEVICES(devicename) FUNCTIONAL_PORT_PROPERTIES (deviceid, ip address) IPINFO (ipname, subnetMask) HARDWARE_PORT_PROPERTIES (mac address)
	UC 13	FUNCTIONAL_DEVICES(devicename, function) FUNCTION
	UC 14	FUNCTIONAL_DEVICES(devicename) FUNCTIONAL_PORT_PROPERTIES (deviceid, ip address) IPINFO IPALIAS
	UC 15	CONNECTIVITY (lkinfo)
	UC 16	FUNCTIONAL_DEVICES(devicename) FUNCTIONAL_PORT_PROPERTIES (deviceid) CONNECTIVTY
PARTITIONING	UC 17	FUNCTIONAL_DEVICES(devicename,system_name) FUNCTIONAL_PORT_PROPERTIES (deviceid) SUBSYSTEM CONNECTIVITY
	UC 18	FUNCTIONAL_DEVICES(devicename,system_name) FUNCTIONAL_PORT_PROPERTIES (deviceid) CONNECTIVITY
	UC 19	FUNCTIONAL_DEVICES(devicename,system_name) FUNCTIONAL_PORT_PROPERTIES (deviceid)

		CONNECTIVITY
EQUIPMENT MANAGEMENT	UC 20	HARDWARE_DEVICES (serialnb, status, location)
	UC 21	HARDWARE_DEVICES (serialnb, status, location)
	UC 22	FUNCTIONAL_DEVICES(devicename) DEVICE_HISTORY
	UC 23	FUNCTIONAL_DEVICES(devicename) HARDWARE_DEVICES (serialnb, status, location) DEVICE_HISTORY
	UC 24	FUNCTIONAL_DEVICES(devicename) HARDWARE_DEVICES (serialnb, status, location) DEVICE_HISTORY
	UC 25	FUNCTIONAL_DEVICES(devicename) HARDWARE_DEVICES (serialnb, status, location) DEVICE_HISTORY
	UC 26	FUNCTIONAL_DEVICES(devicename) HARDWARE_DEVICES (serialnb, status, location) DEVICE_HISTORY
	UC 27	FUNCTIONAL_DEVICES(devicename, location)
	UC 28	FUNCTIONAL_COMPONENTS (cpnt_name,snbid) HARDWARE_COMPONENTS (serialnb, status, location,hwtype) COMPONENT_HISTORY
	UC 29	FUNCTIONAL_COMPONENTS (cpnt_name,snbid) HARDWARE_COMPONENTS (serialnb, status, location) COMPONENT_HISTORY FUNCTIONAL_DEVICES(devicename) HARDWARE_DEVICES (serialnb, status, location) DEVICE_HISTORY
	UC 30	COMPONENT_HISTORY
FAULT DETECTION	UC 31	FUNCTIONAL_DEVICES (devicename, serialnb, location) FUNCTIONAL_PORTS (deviceid, portid) CONNECTIVITY
	UC 32	FUNCTIONAL_DEVICES (devicename, serialnb, location) FUNCTIONAL_COMPONENTS HARDWARE_COMPONENTS FUNCTIONAL_PORTS (deviceid, portid) MICROSCOPIC_CONNECTIVITY CONNECTIVITY
	UC 33	FUNCTIONAL_DEVICES (devicename, serialnb, location) DEVICE_HISTORY
	UC 34	FUNCTIONAL_DEVICES(devicename, serialnb) HARDWARE_DEVICES (serialnb, status, hwtype) DEVICE_HISTORY

Table 15. Verification of the completeness of the schema.

5.8 Conclusion

In this chapter, we have described the CIC DB schema which represents the information about configuration, connectivity and inventory. It has been obtained using the ERM and the use cases defined in Chapter 4. It is the first part of the database layer.

The efficiency of the database schema is essential to implement adaptive tools. A priori, there are some attributes such **CONNECTIVITY.lkweight** or **FUNCTIONAL_DEVICES.node** which can appear unclear in this chapter. In the next chapter, we will present how to automatically generate and update the routing and destination tables and how dynamically get all the paths between two modules. Then the use of these parameters will become more precise. It is the second part of the database layer with its set of PL/SQL routines.

References

[1] DATABASE SYSTEMS, a practical approach to design, implementation, and management written by Thomas Connolly and Carolyn Begg, 2002. Third Edition ADDISON WESLEY. ISBN 0−201−70857−4. 1236 p.

[2] List of requirements and use cases regarding the PVSS configuration framework for LHCb http://lhcb-online.web.cern.ch/lhcb-online/configurationdb/default.htm#Doc.

[3] R. Jacobsson, TFC and Readout Partitioning, LHCb Specification, February, 2006. EDMS 706253 v1.0.

[4] ORACLE, Oracle® Database, SQL Reference 10g Release 2 (10.2), December 2005. p1428. B14200-02. 1428 p.

[5] How many primes are there, http://primes.utm.edu/howmany.shtml#pi_def.

[6] The 1,000 smallest prime numbers, http://www.math.utah.edu/~pa/math/primelist.html.

Chapter 6 Automated creation of routing and destination tables using PL/SQL

This chapter describes the PL/SQL programs, part of the database layer. First, it gives some definitions about how I have modelled the nodes of the connectivity of a subsystem. The connectivity of the LHCb experiment is then represented as an ensemble of the connectivity of the different subsystems. Secondly, it describes the principles of the algorithm I have implemented to create the routing tables. It has been written as a PL/SQL package. Thirdly it presents the creation of the destination table, an extension to the algorithm that I used to handle the TFC partitioning and the automated creation of the dhcp config file. It is one of the key elements to build a set of autonomic tools.

Finally, it presents other PL/SQL programs I had implemented when the SQL statements were too complex and not suitable to be embedded.

6.1 Introduction

6.1.1 Problem

A routing table (for the DAQ switches) or a destination table (for the TFC switch or for the DHCP servers) provides information on how to reach possible destinations. To allow the creation of automatic routing or destination tables, we need to know if a device can be a destination, i.e. if it can receive packets. Typically a PC in the trigger farm will be a possible destination whereas a switch will not be a possible destination.

The query “Give all the paths (in a subsystem) which goes through a given device” is a problem too. As finding the longest path in a graph is a NP complete problem [1], finding all the paths is also a NP complete problem. So there is no algorithm which can solve this problem in a polynomial time, i.e. rapidly. Usually heuristic algorithms are used (tabu search [2] or genetic [3] algorithms for example). In our context, these types of algorithms could not be used as the output of the algorithm must be deterministic, i.e. same output at each execution of the algorithm.

We introduce a parameter M, the maximum path length, i.e. the maximum number of hops to put a limit on the research of paths. In the LHCb context, the topologies were such that it was sufficient to reduce the complexity of the problem. So the problem can be reformulated as finding all the paths whose length is less than M. M is set by the user or the application program.

The execution time depends on the topology of the graph, i.e. the number of vertices and the maximum path length found (the worst case is when it is a fully-connected graph because there are more paths).

The algorithm below has been described in [4].

6.1.2 Intermediate and host nodes and paths

A device can be either an intermediate or a host node. An intermediate node (switches, splitters, and L0 electronics) transfers the data without processing and manipulating it. A host node processes and modifies the data such as TELL1 boards and PCs. A host node has a more complex structure than an intermediate node. For example, the input data of a TELL1 board is generally a digital signal. The output data of a TELL1 board is zero-suppressed and formatted

according to the MEP protocol. Figure 78 shows a slice of the DAQ connectivity. Orange boxes are host nodes (VELO_L1_21 and FARM0101 for instance are host nodes) and non-filled boxes are intermediate nodes such Force 10.

The **FUNCTIONAL_DEVICES.node** column, which is a flag, contains this information. The user must specify if the functional device is a host node (node=1) or an intermediate node (node=0). A host node is also the last device in the subsystem flow. So referring to Figure 78, VELO_L1_21 will have node set to 1 whereas Force 10 will have node set to 0.

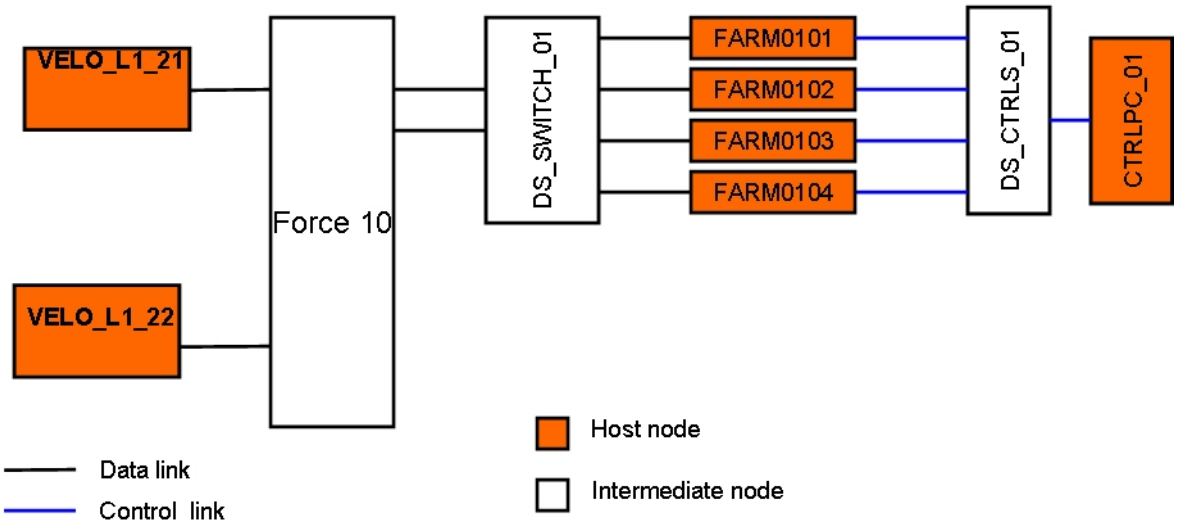


Figure 78. Concept of host and intermediate nodes.

The concept of host and intermediate nodes is very useful to determine whether a device can be a destination. Only a host node can be a destination in a routing and a destination table.

In our context, a path can be defined as a sequence of nodes where there is one host node (the first node is not taken into account) and the last node is a host node. In other words, the pattern *intermediate node - host node - intermediate node* is not allowed in a path. Referring to Figure 78, [DS_SWITCH_01, FARM0101, DS_CTRLS_01] is not an allowed path as there is a host node between two intermediate nodes. A path can contain at most 2 host nodes. The position of a host node in a path is either the starting or terminal node.

The maximum number of hops (M) corresponds to the maximum number of nodes in a path. This parameter is a characteristic of the network.

A routing path is a special path which starts from an intermediate node (Force 10) and ends at a host node (FARM0101 for instance).

This concept of host and intermediate nodes allows splitting the huge connectivity of the whole LHCb experiment into smaller parts which corresponds to the subsystems. It has been applied for all the subsystems (of the detector and Online). Indeed there is no need to look for the RICH connectivity if a user or an application searches paths between a device A and a device B in the VELO subsystem.

6.1.3 Link and path weights

To compute and find the routing paths easier, we have introduced the concept of link and path weights.

The **CONNECTIVITY**.**link_weight** column represents the weight of a link noted *W(L)* and automatically set to (see Figure 79):

- 0 if the link is between 2 intermediate nodes
- 1 if the directed link is between a host node and an intermediate node
- 2 if the directed link is between an intermediate node and a host node.
- 3 if the link is between two host nodes (although not used here)

The path weight *W(P)* is defined as the sum of the link weights along the path. By using the definition of the routing path, we can derive the following theorem which will be used to find the subset of routing paths from paths.

A path P of length J is a routing path of length J

$$\Longleftrightarrow \left(\begin{array}{c} W(P) = \sum_{i=1}^{i=J} W(L)_i = 2 \quad and \\ W(P) = \sum_{i=1}^{i=J-1} W(L)_i = 0 \end{array} \right)$$

where $W(L)_i$ corresponds to the weight of the i^{th} link in the path P. Thus, a path of length J is a routing path of length J if and only if the all the weights of the links (so the $J\text{-}1^{th}$ links) are equal to 0 and the weight of the last link (the J^{th} link) is equal to 2. The proof is given in the Appendix A. Figure 80 shows an example of a routing path.

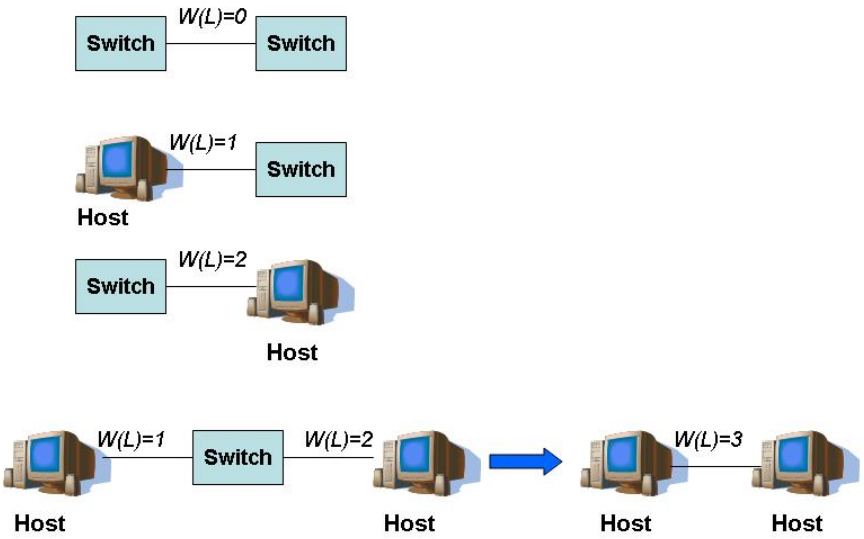


Figure 79. Link weight concept.

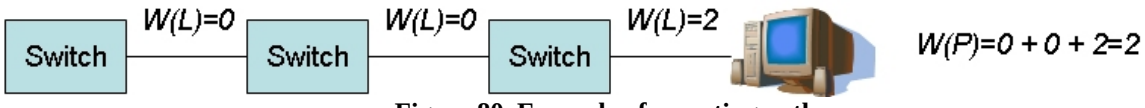


Figure 80. Example of a routing path.

6.2 Algorithm to generate routing tables

6.2.1 Routing tables (reminder)

A routing table consists of providing the following information:

- IP address of the destination;
- Port number to which the IP packet should forwarded;
- IP address of the interface of the next hop;
- Subnet mask of the next hop.

The concept of routing tables has been explained in detail, in Chapter 3, section 4.

6.2.2 Principles of the algorithm

Let us denote L(node i, node j), the link between node i and node j, W(L) the weight of the link and S(node i, node j, node k,…., node t) a sequence of nodes between node i and node t. W(S) is the weight of the sequence (sum over the weight of the links which compose the sequence)
We give an overview of the algorithm which finds routing paths using the connectivity of the DAQ network. The input parameters are the name of the router and the parameter M (default value 10). The steps are as followed:

1. Simplify the connectivity of the system by removing the port level. For instance if there is two links between device A and device B, we just consider it as one link between device A and device B. It is for efficiency reasons.
2. Revert all the links which are bidirectional so that we really find all the paths. For instance if there is a bidirectional link between device A and device B. It is saved as one link starting from device A to device B and one link from device B to device A.
3. Find all the links which starts from the given router and which have a link weight equal to 2. We then found all the routing paths of length 1.
4. Group the links by per of two by making sure that it verifies the four conditions:
    a. the second node of the first link is equal to the first node of the second node (necessary condition to build a path);
    b. the first node of the first link is not equal to the last node of the second link (to avoid cycles);
    c. the weight of the first link is not equal to 2 (to verify the routing path condition where only the last link should have a weight equal to 2).
    d. Compatibility of the link types between the two links to ensure a consistent path. If a link carries data traffic and another link carries controls traffic, the two links cannot be compatible. But if a link carries both controls and data traffics and another link carries only data traffic, then the two links are compatible and the type of this link pair is then data traffic only.

    In other words, the pair of links (which is a sequence of 3 nodes) is valid if and only if S(node i, node j, node k)= L(node i, node j) and L(node j, node k) where node k ≠ node i and W(L(node i, node j))≠2.
5. At each iteration, add a link L(node u, node v) to a sequence of nodes S(node i, node j, …, node t) already found if it verifies the following conditions:
    a. The weight of the sequence is not equal to 2 otherwise the sequence is already a routing path so we don’t touch it.
    b. node u= node t. The first node of the link should correspond to the last node of the sequence of the nodes (otherwise there is no communication between them).

c. node v is not already in the sequence of nodes to prevent cycles.
d. The link type of L(node u, node v) should be compatible with the link type of the sequence of nodes.

If the node passes the conditions, it is added to the sequence of nodes and the weight of the sequence is updated and the type of the sequence too.

So at each iteration, the length of the sequence is increased by one. The loop stops when all the paths found are routing paths, i.e. all the weight of the paths are equal to 2 or the path length is greater than M.

6. Each routing path is now completing with the port numbers.
7. Then for each distinct destination found (last node in the routing path), select the shortest path found.
8. Information about the IP and MAC addresses is performed dynamically (using the port number of the device) when loading the routing table of the given switch to make the updates of the routing tables easier.

6.2.3 Convention

Tables which are suffixed by "_TEMP" are temporary tables. For instance, there is **PATH_LINES** which is a real table and **PATH_LINES_TEMP** which is a temporary table with the same structure as **PATH_LINES**. These tables are not represented in Figure 81 for clarity purposes. An exception has been made for LINK_PAIRS and AGGREGATED_LINKS which are temporary tables, because we estimated that they are important tables. There are no constraints as we do not define constraints for temporary tables. Intermediate results are stored in temporary tables.

6.2.4 Initialization

The input parameters of the routing algorithm are the name of the switch (the one for which we want to generate the routing table) and M..

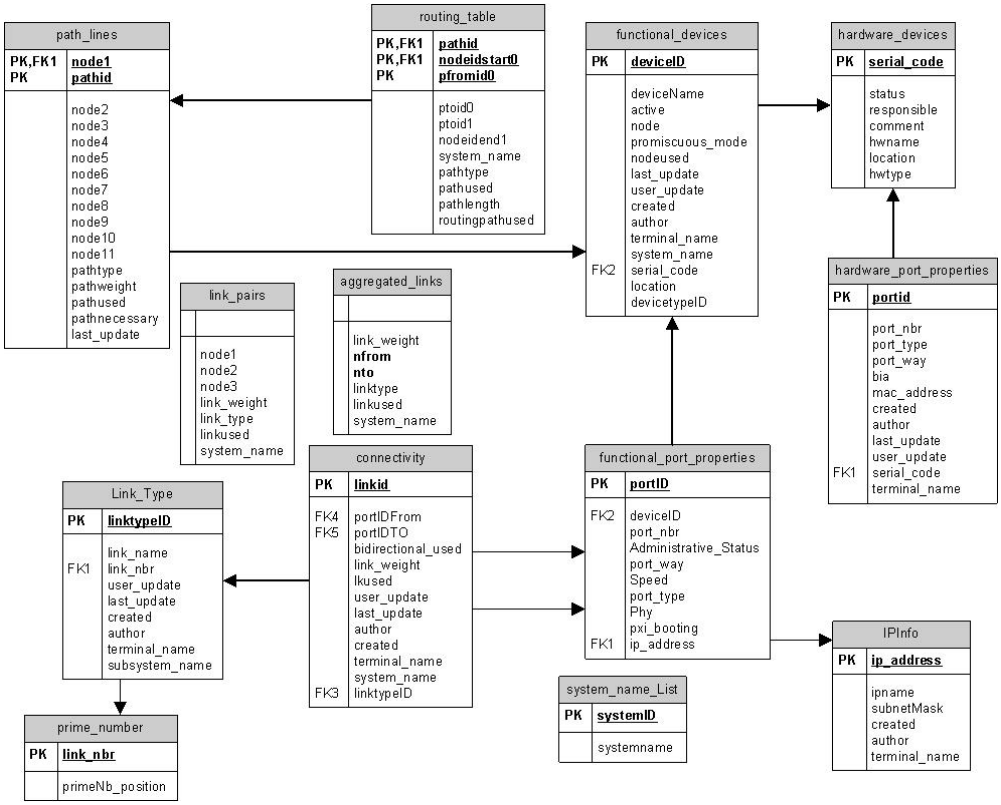


Figure 81. Path modeling.

The algorithm to generate the routing table is based on the following steps:

- Create the **AGGREGATED_LINKS** table (a temporary table)[9] which contains all the links between devices. If a link is bidirectional, we store the reverted link. The principles of this creation are shown in Figure 82. The port number concept is not considered. For instance if the Force 10 router is connected via 10 links to a distribution switch. In the **AGGREGATED_LINKS** table, one link is considered between the Force 10 and a distribution switch. It is derived from the connectivity table (see Figure 81). This step permits to reduce the number of links to be handled.

[9] Temporary tables have no foreign key and no primary key. That is why **LINK_PAIRS** and **AGGREGATED_LINKS** seem disconnect from the schema. They are temporary tables as there is no need to keep their content.

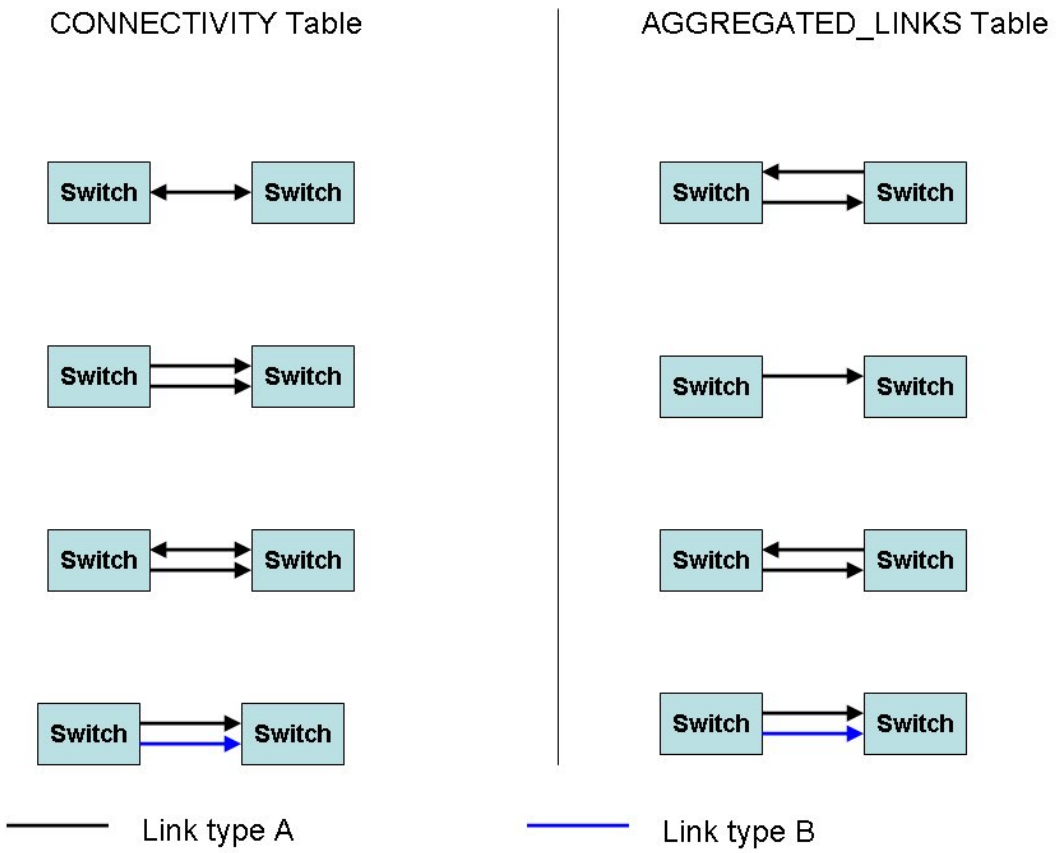


Figure 82. Generating the AGGREGATED_LINKS table using the CONNECTIVITY table.

- Create the **LINK_PAIRS** table (a temporary table) which contains all valid pairs of successive links (one node in common). For instance, the link between Force ten and distribution switch 1 and the link between distribution switch 1 and Farm node 1.

  To create the **LINK_PAIRS** table, we perform a self-join of the **AGGREGATED_LINKS** table with the following constraints:

  - **Link1** is defined by (**Node_1**, **Node_2**) and **Link2** is defined by (**Node_2**, **Node_3**) (referring to Figure 81) where **Node_2** corresponds both to **Node_to** of link1 and to **Node_from** of link2.
  - The **link_weight** of link1 must be equal to 0 because we want to find routing paths (i.e. it starts and ends from/at a switch and, as we are looking for pairs of links, we exclude the switch-host links).
- The **PATH_LINES_TEMP** table is initialized with the elements from the **AGGREGATED_LINKS** (to find path length equal to 1) and **LINK_PAIRS** table which have the switch given as input parameter as a starting node (**Node_1** column).

  We then have found paths which have a length equal to 1 or 2. These paths are inserted in the **PATH_LINES_TEMP** table. If the path length is equal to 1, then the path is inserted as a row into the **PATH_LINES_TEMP** table using the columns **Node_1**, **Node_2**. If the path length is equal to 2, then the path is inserted as a row into the **PATH_LINES_TEMP** table using the columns **Node_1, Node_2, Node_3**.

6.2.5 Body

This subsection explains how we find the routing paths.

We iterate over *i* which represents the path length.

At each iteration of *i*, a join between the **LINK_PAIRS** and the **PATH_LINES_TEMP** tables is executed. This means that a path P, with *W(P)=0* (i.e. having not reached a host) is completed with an element from **LINK_PAIRS** whose first link is equal to the last link of P.

If no such pair exists, the path P is removed. There may be more than one pair which satisfies the conditions. Thus if there are N possible pairs, these N possible pairs will be appended to P and there will be N new paths (i.e. N new rows in the **PATH_LINES_TEMP**).

At the end of iteration *i*, we have found all the paths of length *i* and inserted them in the **PATH_LINES_TEMP** table and we have filled the *i* +1 Node columns of **PATH_LINES_TEMP** table.

For each iteration *i*, the detailed description of the steps is as follows:

1. In the **PATH_LINES_TEMP** table, select the paths P where W(P)=0. (The last column filled is **Node_*i*)**.
2. Find all the possible pairs of links where (**Node_*i-1***, **Node_*i*)** is equal to (**Node_1**, **Node_2**) of **LINK_PAIRS** table and check that there is no cycle (i.e. a node appearing twice in the path).
3. Insert these new valid paths in the **PATH_LINES_TEMP** table. So the **Node_1** to **Node_*i+1*** columns are filled in.
4. Delete the old paths where *W (P)*=0 and **Node_*i+1***=0.
5. Increment *i* by 1.
6. Stop the loop if *i* is greater than M or if all the paths are routing paths, i.e. all paths verify *W (P)*>0.
7. Go back to the port level for the first and last links and insert the **portid**s of the network interface starting the path, ending the first link and ending the path into **ROUTING_TABLE_TEMP**. Finally, we resolve multiple paths to a given (destination, network interface) by setting the **routingpathused** column to 1 for the shortest routing path (required by the DAQ team).
8. Insert the valid routing paths found in **PATH_LINES_TEMP** into **PATH_LINES**, in **ROUTING_TABLE_TEMP** into **ROUTING_TABLE**.

   Commit to delete the content of the temporary tables**,** except the content of **AGGREGATED_LINKS** and **LINK_PAIRS.** They are kept as they can be reused for another switch if it is part of the same subsystem.

This algorithm has been tested against several network architectures including full mesh layouts (see Chapter 9).

Remark on step 6:

If the loop is stopped because of M, paths whose length is greater than M are not found. We trust the user or the application in setting a correct value of M.

6.2.6 Routing table

The **PATH_LINES** table contains all the routing paths of a switch in detail with the different hops. The **pfromid0, ptoid0** and **ptoid1** columns of **ROUTING_TABLE** respectively represent the **portid** of the network interface of the nodeid_start0, the **portid** of the network interface of the next hop and the **portid** of the destination network interface.

The port number to which the packet should be sent is retrieved using **pfromid0.** The IP and MAC addresses of the next hop are found using **ptoid0** and the IP address of the destination is known using **ptoid1.**

The routing paths used to program the switch are stored in the **ROUTING_TABLE** table with **routingpathused** set to 1. It allows a better update and management of paths in case of a problem with a port or a device.

A join between the **ROUTING_TABLE** table, the **FUNCTIONAL_PORT_PROPERTIES** and **IPINFO** tables permits to get the IP address and the subnet mask. We do a join between the **ROUTING_TABLE** table and the **FUNCTIONAL_PORT_PROPERTIES** and **HARDWARE_PORT_PROPERTIES** tables to get the **mac_address**. To avoid many updates in case of a MAC address or an IP address changes, the two joins are performed on the fly, i.e. when the user asks for loading the routing tables.

All the routing tables, i.e. all the routing tables of the DAQ switches, are stored in **ROUTING_TABLE** (one table only).

6.2.7 PL/SQL package

All the steps which have been previously described have been included in a PL/SQL package, **routingtable_pck** (the interface is shown in Appendix B). The package body has 1797 lines of code.

PL/SQL is a proprietary (Oracle) language; the code is executed at the server-side. PL/SQL can be embedded in other languages such as JAVA, C, PERL, etc. A PL/SQL package is stored in its compiled form. The parsing of SQL queries is performed only at compiling time.

When a procedure of a package is called, first Oracle gets the package and loads it into memory if it is not already there. So performance is improved as parsing SQL queries can be quite time consuming depending on the complexity of the query.

By using PL/SQL one avoids overloading of the network by very long sequences of SQL queries. Also the maintenance of the routing tables is easier. Whenever there is a change in the **CONNECTIVITY** TABLE, **ROUTING_TABLE** and **DESTINATION_TABLE** related to DAQ or TFC system are recreated.

Generating a routing table is performed using 4 functions of **routingtable_pck**.

1. The first function creates and filled the **AGGREGATED_LINKS** and **LINK_PAIRS** tables.
2. The second function finds all the routing paths which start from the given devices using the logical view. These are stored in the **PATH_LINE_TEMP** table. **STARTEND_TEMP** is also filled with the two first and the two last nodes. It will be used to select the right port interfaces.
3. The third function maps the start (the first link) and the end (last link) of the path with **PORT_PROPERTIES.portid** with all the checks (same link type, bidirectional link used, link used or not) and inserts them in **ROUTING_TABLE_TEMP**. One routing path is selected by (destination, network interface) among the valid paths, i.e. where no link is disabled or broken. Set **routingpathused**=1 to the selected routing paths.
4. The fourth function deletes the old entry related to the given switch and inserts all the results in the tables **PATH_LINES** and **ROUTING_TABLE**.

6.2.8 Completeness of the algorithm

The routing algorithm finds all the paths less than M (less than 10 in the context of LHCb). The proof relies on the “join” operator reliability. Figure 83 illustrates the concept. If a valid path is not found with a length less than M, it means that during the join operation the code could not find a pairs of links which matches the current path. It means that this pair of links

is missing. So it means that this pair of links is not in the LINK_PAIRS table which is the result of a self-join with constraints of the AGGREGATED LINKS TABLE. The join operator in SQL is known to be reliable. So if this pair of links is not in the table, it means that the pair of links fails to satisfy the constraints, which is in contradiction with the fact that is a valid path.

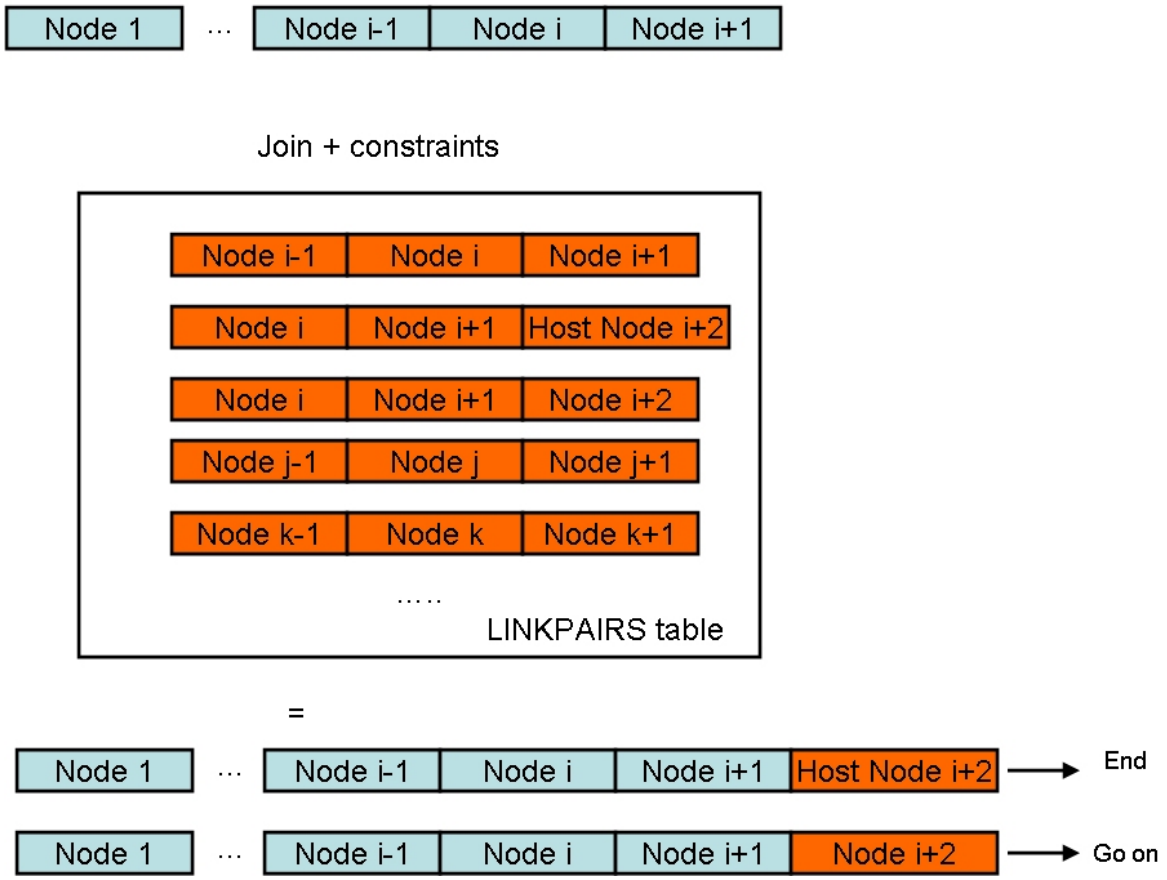


Figure 83. Concept of finding the paths. The path starting from Node 1 to Host Node i+2 is a routing path. The other path ending at Node i+2 is still not finished, we go on if the i+3<M.

6.3 Extensions of the routing table algorithm

6.3.1 Partitioning

6.3.1.1 Destination table

In Chapter 2, section 2.3, partitioning has been handled using the destination of the TFC switch. A destination table of a device consists of the following columns (which are almost similar to **ROUTING_TABLE**):

- **Deviceid of the node1** (which starts the path). This node corresponds to the **deviceid** of the functional device for which the destination table is generated. For instance, if we generate the destination table of the TFC switch, the value of this column corresponds to the **deviceid** of the TFC switch.
- **Portid of the node1** (from which the link starts). This column permits retrieving information about the port which starts the path. In the case of the TFC switch, it corresponds to the port IDs of the output ports.

- **Portid of the last node** (at which the link ends) allows retrieving information about the port which ends the path. In the case of the TFC switch, it corresponds to the port IDs of the input ports of the destination devices, i.e. the TELL1 boards.
- **Deviceid of the destination devices**.
- **System name** which corresponds to the list of subsystems of which the destination device is part.
- **Pathused** which indicates if the path is functional (1) or not (0). For instance if a device is broken and not replaced or if a device needs to be excluded for debugging reasons, for instance, all the paths which go through that device are disabled, i.e. **pathused**=0.

All the destination tables which are generated are stored in **DESTINATION_TABLE**.

6.3.1.2 Algorithm principles

The routing algorithm has been adapted to generate a destination table as the concept is similar. The main difference is that the destination table can be generated for a host node. So the computation of the path weight is slightly different. However the algorithm principles are the same in both cases.

1. The first step is to determine whether the destination table is for a host or intermediate node.
2. If it is a host node, the path weight should be equal to 3 =(1 +2).
3. If it is an intermediate node, the path weight should be equal to 2 = (0+2). It is the same as for the routing tables.

Also all the paths are inserted. In other words, it is possible to have several paths which start from the same pair (**deviceid** of the first node, **portid**) unlike the routing tables.
The functions which are used to generate the destination tables are also included in the **routingtable_pck** PL/SQL package.

6.3.1.3 Example of the TFC switch

Figure 11, in Chapter 1, shows the connectivity of the TFC. The readout supervisors (ODIN) and the TELL1 boards are respectively the sources and destinations. Thus they are host nodes, the other devices are intermediate nodes.

All the links in the TFC system are unidirectional. The TFC switch can only send information to TELL1 boards, which are the only possible destinations in that case.

A readout supervisor can not be a destination as it is a source (it sends information but does not receive any data via the TFC switch). The destination table of the TFC switch will contain around 350 distinct destinations (equal to the number of TELL1 boards).

The TFC system needs to know the output ports of the TFC switch which drive the subsystems in the partition to configure the switch.

Consider the following example. An extract of the TFC destination table is shown in Table 16.

TFC output port nb	Port way	Port type	Destination device name	System ID of the destination
0	**2**	**None**	**VELO_L1FE_04_00**	**1155**
0	**2**	**None**	**VELO_L1FE_04_01**	**1155**
1	**2**	**None**	**VELO_L1FE_07_04**	**1365**
1	**2**	**None**	**VELO_L1FE_07_04**	**1365**
2	**2**	**None**	**RICH1_L1FE_09_08**	**4845**
2	**2**	**None**	**RICH1_L1FE_08_02**	**4845**
3	**2**	**None**	**RICH2_L1FE_00_01**	**5865**
3	**2**	**None**	**RICH2_L1FE_00_02**	**5865**
10	2	None	L0MUON_L1FE_00_03	98355
10	2	None	L0MUON_L1FE_00_01	98355
10	2	None	L0MUON_L1FE_00_02	98355
4	2	None	IT_L1FE_09_00	26445
4	2	None	IT_L1FE_09_00	26445
4	2	None	IT_L1FE_09_00	26445
5	2	None	TT_L1FE_03_01	28905
5	2	None	TT_L1FE_03_01	28905
5	2	None	TT_L1FE_03_01	28905
6	**2**	**None**	**OT_L1FE_00_12**	**13485**
6	**2**	**None**	**OT_L1FE_00_11**	**13485**
6	**2**	**None**	**OT_L1FE_00_10**	**13485**
7	2	None	OT_L1FE_02_05	16095
7	2	None	OT_L1FE_01_21	16095
7	2	None	OT_L1FE_01_09	16095
8	2	None	PRS_L1FE_03_00	915
8	2	None	PRS_L1FE_01_00	915
8	2	None	ECAL_L1FE_02_00	795
9	2	None	ECAL_L1FE_04_00	795
9	2	None	ECAL_L1FE_03_00	795
11	2	None	HCAL_L1FE_01_01	885
11	2	None	HCAL_L1FE_01_00	885
12	2	None	MUON_L1FE_01_00	71355
12	2	None	MUON_L1FE_01_00	71355
13	2	None	MUON_L1FE_01_00	73365
13	2	None	MUON_L1FE_07_00	73365
13	2	None	MUON_L1FE_07_00	73365
14	2	None	L0CALO_L1FE_00_00	105465
14	2	None	L0CALO _L1FE_00_01	105465
14	2	None	L0CALO _L1FE_00_02	105465
15	2	None	L0DU _L1FE_00_01	119685
15	2	None	L0DU _L1FE_00_02	119685
15	2	None	L0DU _L1FE_00_03	119685

Table 16. Extract of the TFC destination table.

For example, for a partition consisting of {VELO, RICH, OT_A}, the SQL query:

```
select distinct port nb from DESTINATION_TABLE r, FUNCTIONAL_DEVICES t,
SUBSYSTEM_LIST l where r.nodeid_start=t.deviceid and
t.devicename='TFC_SWITCH' and l.system_name='VELO' and mod (r.systemid,
l.systemid) =0.
```

will only select the destination devices belonging to these subsystems. The result of the query is 0 and 1.

The same query is performed for RICH and OT_A so that each subsystem can be properly associated with 1 or two output ports of the TFC switch.
The selected destinations with their associated output port(s) are written in bold in Table 16.

6.3.2 Generating the DCHP config file

The destination table is generated for the DHCP server to get all the host nodes which can get their IP address from the given DHCP server.

It has been integrated in a Perl script which is described in the next chapter.

6.4 Other PL/SQL programs

Besides the PL/SQL package **routingtable_pck**, there are other PL/SQL programs which have been implemented to avoid embedding long sequences of SQL queries (essentially updates and insertions).

- **UpdateDeviceHistory** : a PL/SQL function which updates the history table in case the user deletes a functional or a hardware device further to mistypes.
- **UpdateBoardComponent**: a PL/SQL function which updates the status of the microscopic devices (components of the board) according to the change of the status of the motherboard. (the status should be different from TEST).
- **SwapTwoDevices**: a PL/SQL function which swaps two devices and checks that this operation is allowed (for instance same number of ports connected and same device type).
- **InsertSubsystem**: A PL/SQL function which inserts a new subsystem and attributes it a systemID.
- **ComposeFunctionID**: a PL/SQL function which returns the function ID given a list of function names.
- **DecomposeFunctionID**: a PL/SQL function which returns the function names given a function ID.
- **InsertIPaddress**: a PL/SQL function which inserts an IP address. It is used when inserting a port of a device. There is a need to know if the IP address already exists or not. If not, it inserts it.
- **UpdateIPaddress**: a PL/SQL function which updates an IP address further to a mistype or a change. It performs the update for IPINFO, IPALIAS and PORT_PROPERTIES tables.
- **TestUseBoardCpnt**: a PL/SQL function which sets the status of the microscopic devices (components of the board) to TEST when the motherboard goes to TEST.
- **InsertTestBoard**: a PL/SQL procedure which inserts a test board. The name of a board test is automatically generated by the CIC DB.
- **CreateTableSchema**: a PL/SQL function which creates the CIC DB schema (tables, indexes and constraints).
- **DropTableSchema**: a PL/SQL function which drops the current CIC DB schema.

6.5 Conclusion

This chapter describes how routing and destination tables can be generated using the information stored in the CIC DB. The algorithm is based on two main concepts:

- Intermediate and host nodes to make the distinction between possible destinations for the routing table and also to put boundaries to the network of a subsystem
- Paths and path weights to compute the correct paths by avoiding cycles.

The algorithm implemented in PL/SQL is used mainly for fixed queries such as:

- generating the routing tables of the DAQ switches;
- generating the destination table of the TFC switch for partitioning;
- generating the destination table of the DHCP servers to create the dhcp config file.

All the routing tables (DAQ switches) and destination tables (TFC switch and DHCP servers) are respectively in the **ROUTING_TABLE** and **DESTINATION_TABLE**. They are automatically updated.

It also describes the other PL/SQL codes which have been implemented when a query was too complex or implies too many checks. It was the case for some insertions and updates.

So the implementation of the database layer has been described over Chapter 5 and 6. The next chapter will focus on the object layer.

References

[1] M.R.Garey and D.S.Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, 1990. W. H. Freeman & Co. New York, NY, USA. 0716710455. 338 p.

[2] Gendreau M. An introduction to tabu search. In: Glover F, Kochenberger GA, editors. Handbook of metaheuristics. Boston, Dordrecht, London: Kluwer Academic Publishers; 2003. p. 37-54.

[3] Rudolph, S. On a Generic Algorithm for the selection of optimally generalizing neural network topologies. Proceedings of the 2nd International Conference on Adaptive Computing in Engineering Design and Control'96. I.C. Parmee (ed.), University of Plymouth, March 26th-28th, Plymouth, United Kingdom, 79-86, 1996.

[4] L.Abadie, Configuring the LHCb Redaout Network using a database, August 2006.
IEEE Transactions on Nuclear Science, June 2006, Vol. 53, number 3, Part I of three parts. 14th Conference on real time (RT 2005) Stockholm, Sweden, June 4-10, 2005. p995-1001.

Chapter 7 Implementation of the object layer

This chapter describes the object layer which consists of a set of libraries which helps in building autonomic tools to interact with the CIC DB. The first section presents the two Perl scripts I wrote to generate config files for the DHCP and DNS servers. I also present an example of autonomics setup to configure the DHCP and DNS further to a change in the DAQ network. The second part explains the implementation details of the CIC_DB_lib library, namely its API, its structure and its features. It is the core of the object layer as it provides functions to manipulate the information about connectivity and inventory/history. I implemented the library in such way that autonomics tools can be built on top of it by making it intelligent. The third part describes the implementation of the two bindings of CIC_DB_lib. I also discuss the issues during the implementation. Finally the last part gives an overview of the PVSS library (provided by the CERN PVSS Support group) built for recipes.

7.1 Use of Perl scripts to generate config files

7.1.1 DHCP config file

This section describes how the dhcp config file can be generated using the connectivity information stored in the CIC DB. This method will be used to configure the controls network interfaces of PCs in the farm, TELL1 boards and readout supervisors. The IP addresses for the data network interfaces will be attributed differently (but it is not covered in this thesis as it is not fixed yet).

7.1.1.1 Methodology

To create the dhcp config file, we use the following method [1]:

- **Get the host nodes** by generating the destination table of the given dhcp server name. Using the destination table (especially the **portid** of the last node) as described in Chapter 2 section 2.2.2.8, one can retrieve the **PORT PROPERTIES**, **HW PORT PRORPERTIES** and **IPINFO** tables, containing network information (MAC, IP and subnet addresses).
- **Get the boot images**. The boot image information is usually linked to the device type (CPU architecture, kernel version for instance). In most cases, all the farm nodes will have the same boot image, same remarks for the TELL1 boards (per subsystem) and the readout supervisors. However it may occur that a host has its specific boot image. To get the right boot image, we perform the 2 following steps:
    - Step 1: look if the given host name has a specific boot image in the **DEVICE_BOOTING_TABLE** in the CIC DB. If we find an entry corresponding to this node, we select it. If not, go to step 2.
    - Step 2: attribute it the boot image associated with the node's devicetype. We get it from **DEVICETYPE_BOOTING_TABLE**.
- **Getting the subnet ID**. In the CIC DB, the subnet mask is stored instead of the subnet ID. However, one can compute the subnet ID given the subnet mask and the IP address.

For example:

IP address: 137.192.25.15 and **Subnet Mask**: 255.255.255.0.

1. Convert the IP address and the Subnet Mask to binary formats.

IP address: 100001001.11000000.00011001.00001111
Subnet Mask: 11111111. 11111111.11111111.000000000
We consider these two numbers as 2 vectors (32x1).

2. Perform a point wise[10] multiplication of the 2 vectors.

The subnet ID is then equal to 100001001.11000000.00011001.00000000

3. Convert it to decimal format : 137.192.25.0.

Group the hosts by subnet IDs as follows:

```
subnet 137.192.25.0 netmask 255.255.255.0 {
group {
    host pctest32 {
    hardware ethernet 00:00:A2:11:25:B4;
    fixed-address 137.192.25.15;
    filename "lynx_boot_23.nbi";
    }
    host { ... }
    } }
subnet ...
```

At this stage, we have all the necessary information to build the dhcp config file. The next subsection will focus on the implementation.

7.1.1.2 Generating and formatting the dhcp config file

[10] Point wise multiplication is as follows. Consider two vectors *v*(x,y) and *u*(x',y'). The point wise multiplication of vectors *v* and *u* is another vector *z*(xx',yy').

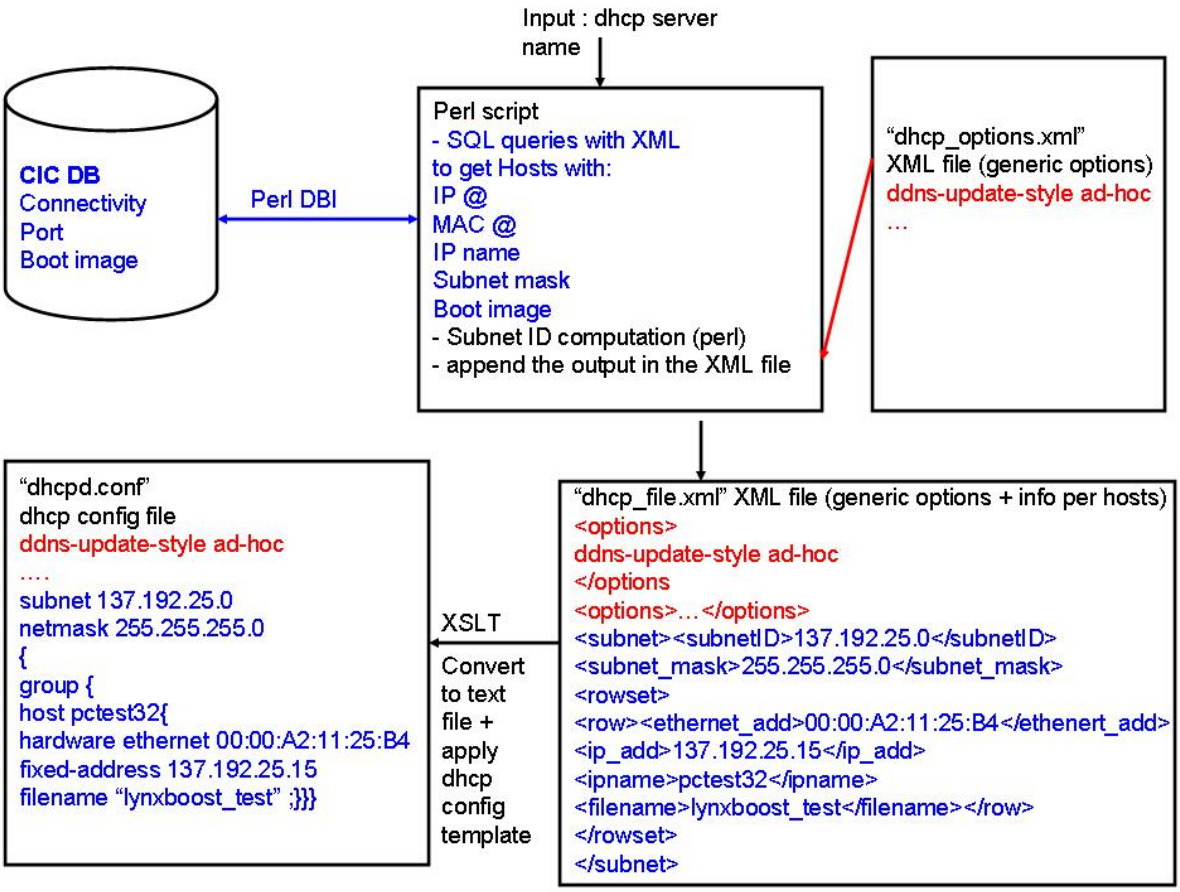


Figure 84. Implementation principles.

Figure 84 describes how the DHCP config file has been generated.

It is based on a Perl script. Perl is one of the most used languages for writing Linux scripts. Perl also includes a lot of packages (XSLT, DBI, etc…) and is very convenient for string manipulation. Also I am more familiar with Perl than with Python. For security reasons, SQL statements have been embedded in the Perl script directly. We did not want to include functions in the CIC_DB_lib (as mentioned in Chapter 3) which allow creating DHCP and DNS config files and make it accessible to anybody. These two Perl scripts will be used only by the DAQ network system administrators and they agree with this method and the choice of Perl.

Any time there is a change in the configuration of the network, the following steps should be carried out:

- The generic options are put in the "dhcp_options.xml" file provided with the application. There should one option per line. Referring to Figure 84, the generic options are written in red.
- Execute the Perl script, with the dhcp server name as input argument. It is case-sensitive. The Perl script performs three steps.
    - It obtains the network information (IP and MAC addresses, subnet_mask and boot image) from the CIC DB using Oracle XML[2] features such as "xmlelement".
    - It writes the results in the XML file "dhcp_file.xml". The previously defined generic options are printed at the beginning of

"dhcp_file.xml". The result of the query is encapsulated in XML tags. The results are ordered by subnet ID.

- It generates the dhcp config file using XSLT. XSLT is used to convert the XML file obtained into a dhcp config file. To do so, there is an XSL file which reads the XML file. It converts the XML tags into words which are understandable by a dhcp server.

  For instance, the XML tag <ethernet_add> is converted into hardware ethernet, <ipname> to host, etc.

The output of these operations is the dhcpd.conf. All the files created and used are located in the same directory. The dhcpd.conf is copied in /etc/... manually by the network administrator. It is for security reasons. The Perl script "dhcpCfg_generate.pl" can be found in Appendix C.

7.1.1.3 Excluding nodes

If for any reason, a host needs to be excluded from configuration by a given dhcp server, it can be disabled by setting **FUNCTIONAL_DEVICES.nodeused** to 0. This function will exclude all the links affected by this change (**CONNECTIVITY.lkused** set to 0) and as a consequence, will update the **DESTINATION_TABLE.pathused** to 0 if one link has been disabled in a path.

When a host is disabled in the table, the Perl script "dhcpCfg_generate.pl" needs to be rerun.

Then the hosts should be included back again. This has to be done because the host was disabled for generating a correct DHCP config file for a given DHCP server.

The following example illustrates this.

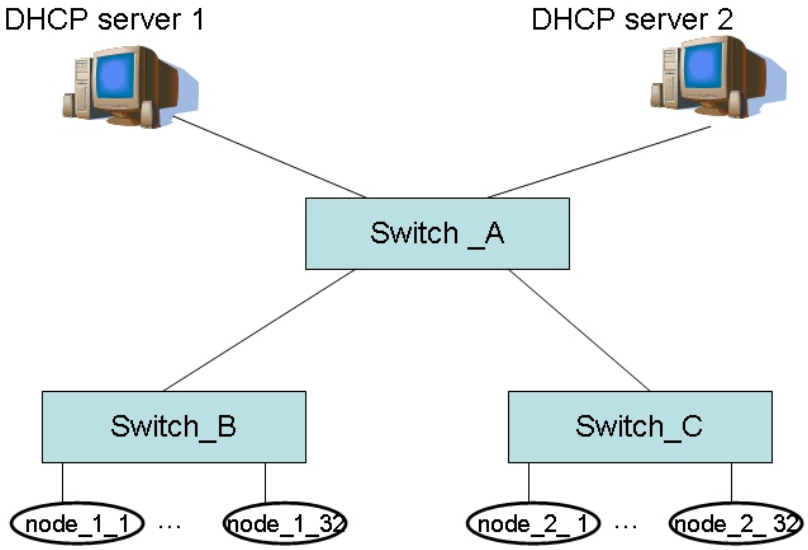


Figure 85. Example of a topology where it is mandatory to exclude nodes.

Assume a connectivity situation as in Figure 85. There are two DHCP servers, DHCP server 1 and DHCP server 2. The DHCP server 1 configures nodes from **node_1_1** to **node_1_32** and DHCP server 2 configures nodes from **node_2_1** to **node_2_32**.

Both DHCP servers 1 and 2 could configure all the nodes. Therefore the destination table of DHCP server 1 is the same as that of DHCP server 2. It will contain all the host nodes, i.e. from **node_1_1** to **node_1_32** and from **node_2_1** to **node_2_32**. So

the destination table for both DHCP servers 1 and 2 will contain too many reachable hosts. Consequently the generated dhcp config file may be wrong depending on the network policy[11]. So it is important to provide a solution if this case occurs. Then it will be up to the network administrator to decide what to do.

This problem can be solved by disabling hosts. The destination table of DHCP server 1 should contain only hosts from **node_1_1** to **node_1_32**. Hosts from **node_2_1** to **node_2_32** must be excluded. When they have been excluded the Perl script "dhcpCfg_generate.pl" should be executed to generate the dhcp config file for DHCP server 1. Then the excluded nodes must be included back so that the dhcp config file for DHCP server 2 can be generated. An example of the C code[12] is in Appendix D. It shows how this case can be handled.

In the same way, links can be disabled by updating **CONNECTIVITY.lkused** column.

In the previous example, another way to do could have been to exclude the link respectively between switch A and switch B (respectively switch A and C) to generate the dhcp config file for DHCP server 2 (resp. DHCP server 1).

7.1.1.4 Including nodes

The DAQ farm will grow over time as more PCs will be added. How will the generation of the dhcp config file be affected by the arrival of new PCs or new TELL1 boards?

The impact is rather slight as the new devices and their connectivity only have to be added to the CIC DB. The Perl script "dhcpCfg_generate.pl" should be rerun. If the insertion is done from PVSS, "dhcpCfg_generate.pl" can be automatically executed using a PVSS script.

7.1.1.5 Autonomics set up

The use of autonomics principles is reflected through 7.1.1.3 Excluding nodes and 7.1.1.4 Including nodes. The human intervention is reduced. Using the following setup, most of the reconfiguration steps are automated as illustrated by Figure 86. The user makes the changes on the DAQ network such as adding new farm PCs or updating IP addresses using PVSS panels. Then all the changes are saved in the CIC DB using the PVSS extension of CIC_DB_lib. It will dynamically update the routing and destination tables if needed. If the changes are successful the PVSS panel asks for the list of DHCP servers to the CIC DB. Then using DIM and this list, the Perl script "dhcpCfg_generate.pl" is automatically executed on each DHCP server.

So in this setup, the user did not have to recreate the routing tables or to rerun the "dhcpCfg_generate.pl manually.

[11] Depending on the network setup, it does not matter if the dhcp config file contains more hosts than needed as the DHCP server will never get a BOOT request from them.

[12] It is written in C because it calls functions part of the CIC_DB_lib, namely to disable and enable the hosts)

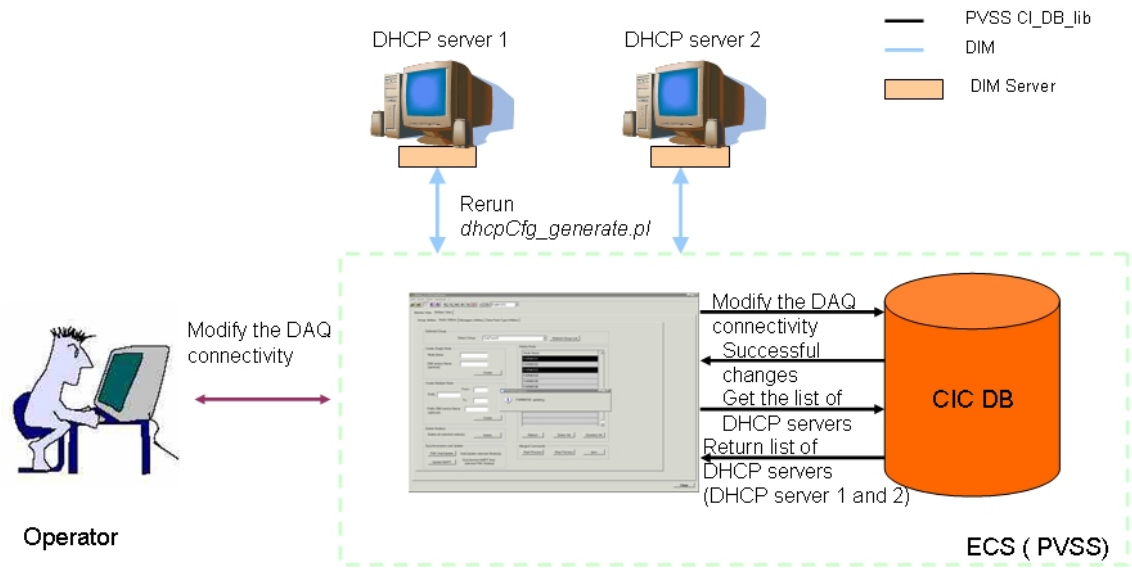


Figure 86. An autonomic setup to update the dhcp config file using PVSS.

7.1.2 DNS files

The DNS provides the correspondence between IP address and IP name. Its principles have been described in Chapter 2 section 2.2.2.9. We have tried to adopt a similar approach as for the DHCP config file. The information which needs to be predefined is the name of the domain, the name of the authoritative DNS server and the maximal number of times that a DNS config file can be recreated in one day. They are pretty static so they are saved as global variables in the script.

To create the DNS files, we have split the code into 2 parts, one part which generates the forwarding file (given an IP name, retrieve the IP address) and the second part which generates the reverse file (given an IP address, get the IP name). The whole Perl script "dns_generate.pl" can be found in Appendix E.

7.1.2.1 Outline of creating the DNS forwarding file

This subsection describes how the DNS forwarding file is generated. The following steps have been performed:

- Get the next serial. The serial identifies uniquely the dns file. Its value should be the same for the two set of files (forwarding and reversing as presented in Chapter 2 section 2.2.2.9). The serial is obtained by concatenating the current date (year/month/day) and a number. This number starts with 0 and is incremented whenever the DNS set of files has been re generated during a same day. In the example shown below the serial is equal to 200607130. 2006 is the year, 07 is the month, 13 is the day. Then the last digit is 0 (it is the number which means that it has been created only once and it was on July, 13th 2006). This number should be less than P, maximum number to generate the DNS set of files (forward and reverse files), fixed by the network administrator. The default value (common value) is 9. If the serial is invalid, the program exits (and does not go through the reverse part).

  The two types of files start with some generic options showed below:

```
$TTL    86400; minimum TTL (time to live) in seconds as of bind
8.2
```

```
# name of the domain "." is important              name of the DNS
server
ecs.lhcb.       IN      SOA     dns01.ecs.lhcb.    root.localhost.
(
#some generic options stored in the XML generic options file
                                          200607130 ; serial
                                              3h ; refresh
                                            3600 ; retry
                                              4w ; expire
                                            3600 ; ttl
                                          )
```

- Use of XML in the SQL queries to get results formatted as followed: <row><ipname>value</ipname><ipadd>value</ipadd><function>{NS,A,CNAME}</function></row>. Using regular expressions, the domain name is taken off from the ipname.
- Use of XSLT code to convert the XML file and the XML generic option file.

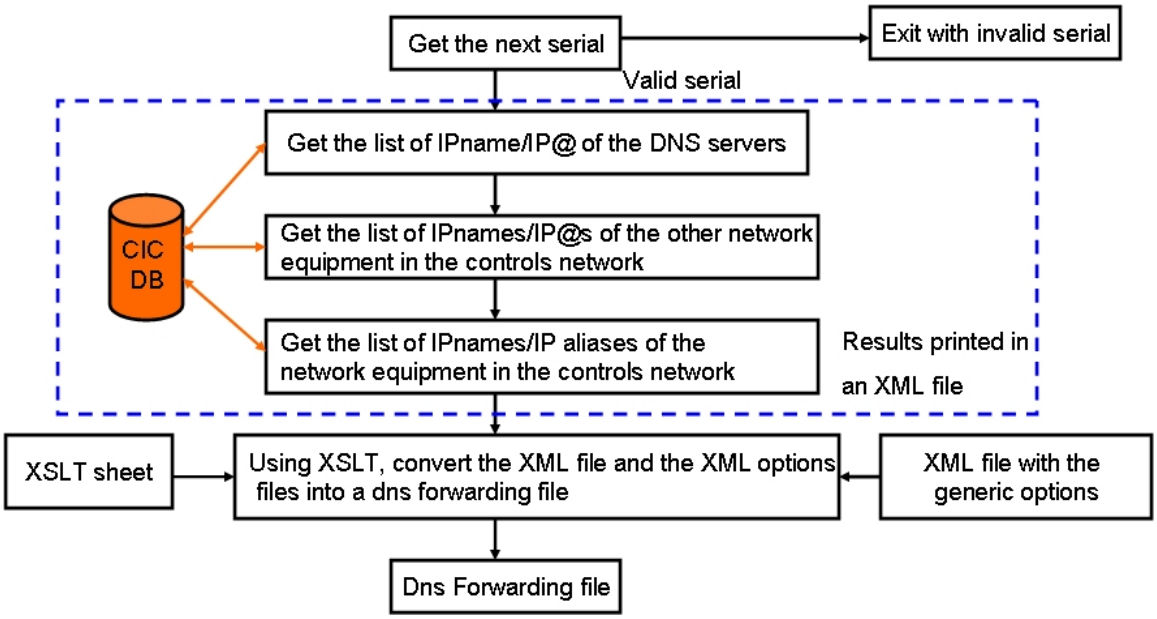


Figure 87. The principles of creating the DNS forwarding file. @ stands for address.

Figure 87 describes the previous steps performed to create the DNS forwarding file.

7.1.2.2 Outline of creating the DNS reversing file

This subsection describes how the DNS reversing file is generated.

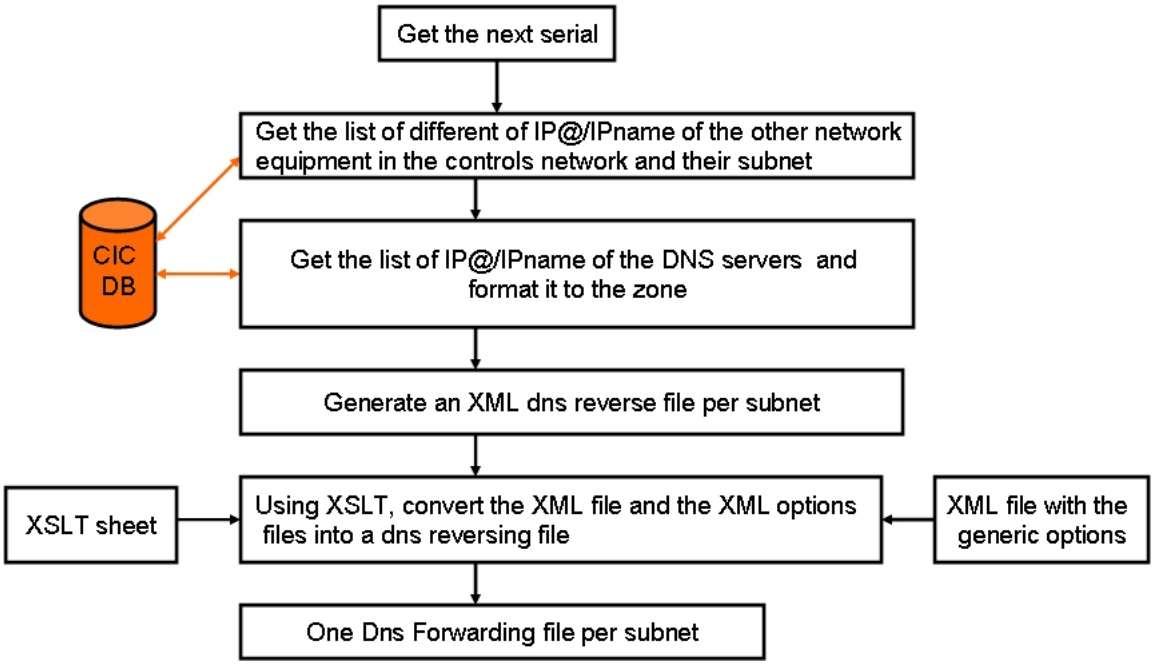


Figure 88. Implementation guidelines of the creating the dns reversing file. @ stands for address.

Figure 88 presents the implementation guidelines of creating the dns reversing file.

- Get the value of the serial parameter. It is equal to the one in the dns forwarding file and it is passed as an input parameter.
- Get the list of the IP addresses and IP names of all the equipment part of the controls network with their subnet ID, using XML embedded in SQL. Store the results in an array.
- For each different subnetID, get the list of IP addresses and IP names of all the DNS servers (even if they are not in the given subnet) and format it for the subnet ID, like:

  ```
  <row><ipadd>05.100.60.137.</ipadd>
  <ipname>DAQ_CTRLPC_60_01.ecs.lhcb.</ipname>
  <function>NS</function> </row>
  ```

  The IP addresses should be reverted. For instance, if the IP address is 123.23.56.45, it becomes 45.56.23.123. A dot should be put at the end to prevent from appending the subnetID at the end of IP address.

  For the authoritative DNS server or also master DNS server (static variable in the script), it is formatted as follows:

  ```
  <row><ipadd>137.56.in-addr.arpa.</ipadd>
  <ipname>DAQ_CTRLPC_10_01.ecs.lhcb.</ipname><function>NS</function
  ></row>
  ```

  In the previous example, 137.56.0.0 is the subnet ID. The “0” is taken off and replaced by “in-add.arpa”. Between <ipname> and </ipname>, there is the full name of the master DNS with the zone name (ecs.lhcb). The dot at the end is essential. If it is omitted, the zone name is appended to the name.

  Add all the IP addresses and IP names which are in the given subnetID.

  Using XSLT and the xml generic options (same as the previous one), convert the xml file into a reverse dns file for the given subnet ID.

The last step is iterated for all the subnet IDs. In the case of LHCb controls network, there are 4 subnets.

7.1.2.3 Autonomics setup

When new PCs are added, they will obtain IP addresses, IP names and eventually IP aliases. The Perl script "dns_generate.pl" must be rerun to add these new entries (IP address, IP name) in the set of dns files. The same problem occurs when new network equipment are no longer used. However there is also the dhcp config file which needs to be updated too. So the Figure 89 suggests an autonomics setup which updates both the dhcp config file and the dns files further to a change in the DAQ network using PVSS. The user modifies the DAQ network setup using PVSS. Then automatically, the DHCP and the DNS servers are reconfigured according to the new setup.

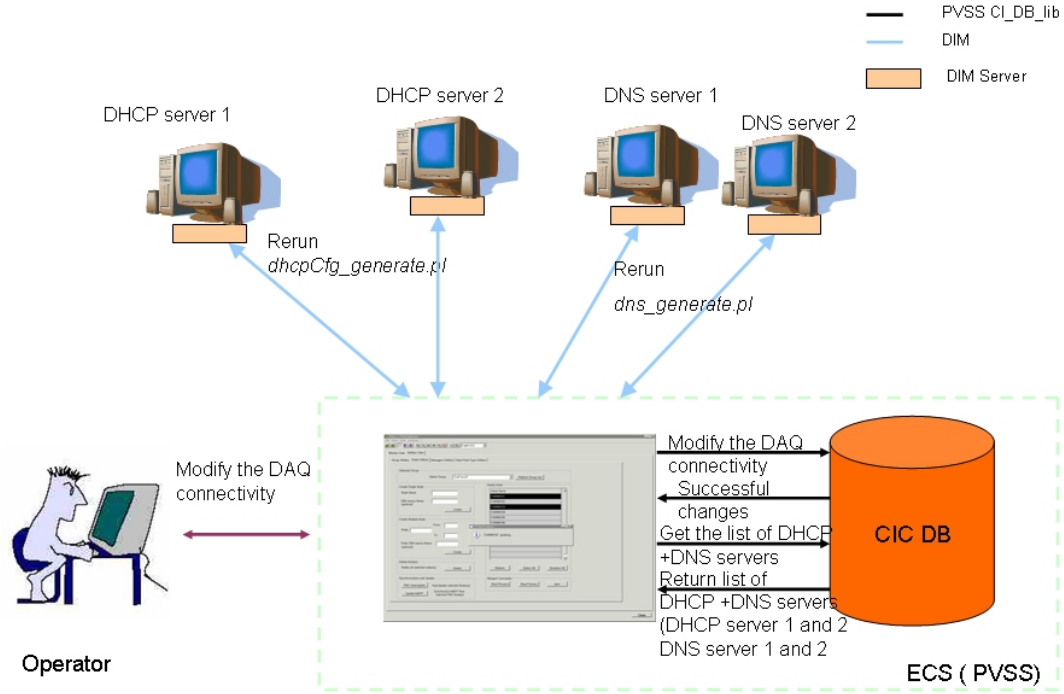


Figure 89. Autonomics setup for the configuration of the DNS and DHCP servers further to a change in the DAQ network setup.

7.2 CIC_DB_lib, a C-library to query the CIC DB

7.2.1 Implementation guidelines

7.2.1.1 The CIC_DB_lib API

The purpose of the API is to allow a non DB expert user to interact with the CIC DB in a safe mode and without any knowledge of the table schema. So the database aspects are hidden from the applications. It also provides a standard interface to the CIC DB. To guarantee database integrity, all the functions of the API are based on DML statements [2], i.e. the non DB-expert user is not allowed to drop a table as a table can be used by more than one application in the LHCb experiment.

To ensure that the library will provide all the information required by the different users of the CIC DB, the first step was to design the API.

The API is split into four parts (see Appendix F for the C interface):

1. functions to query the content of the CIC DB (based on the SELECT statement);

2. functions to populate the CIC DB(based on the INSERT statement);
3. functions to update information stored in the CIC DB (based on the UPDATE statement);
4. functions to delete information stored in the CIC DB (based on the DELETE statement);

The API has been built using the use cases defined in Chapter 4 section 4.3. Then it has been improved and completed through discussions with the users of the CIC DB. As agreed within the LHCb collaboration, anybody should be able to have access to and to use any functions. There are no users' privileges. All the SQL statements are hidden as the user is a non DB expert.

7.2.1.2 Use of OCI

CIC_DB_lib is based on OCI (Oracle Call Interface) [3] to access the CIC DB and on C as the programming language. The main advantage of the OCI interface is that it provides a lot of functions to interact with a database. It is faster than other interfaces (e.g. OCCI) [4] as these are built on top of OCI. It is more stable. It is recommended by Oracle for access to the database. So any type of statement can be done using OCI. However it is a quite complex interface. An example of how to OCI is used is shown in Appendix G.

7.2.1.3 Output format of a SELECT query

The return value of functions based on SELECT statements is formatted as follows (the LHCb Online group has agreed with this format).

- The result of the SELECT is known to be one single row, then the return value is formatted as follows: |column_name: column_value (column_type) | …|, where | is a delimiter. It includes all the functions which return a row of a table given the primary key (e.g. deviceid) or another candidate key (devicename). For example, **`GetFunctionalDeviceTypeRow`** returns the row of a given devicetype. The main advantage of this format is that the signature of this type of functions is the same after adding or dropping one column in the table.
- The result of the SELECT returns a group of rows. The return value is either an array of int which correspond to the primary key of the table such as **`GetPortIDPerDevID`** which returns all the portids of a given deviceid. Or it can be a list of elements formatted as follows. Each row of the list is separated by '\0' so that the mapping into vector of strings is easier (for PVSS and Python).

7.2.1.4 Use of a memory cache for INSERT and UPDATE

For INSERT and UPDATE statements, a cache has been implemented to allow users to insert and update many rows in one go (bulk collect feature). It is well known that it is faster to insert multiple rows in one go than one row at the time.

To do so, a cache (using buffers) has been implemented to store all the rows which need to be inserted or updated. The user has to set the parameter **last_rows** (input parameter in the insert and updates functions) to 1 to indicate that it is the last row which will be inserted or updated. So as long as this parameter is equal to 0, nothing

will be inserted nor updated in the CIC DB. When this parameter is set to 1, all the rows are stored using **OCIBindArrayOfStruct** and sent to the CIC DB.

When inserting, deleting or updating many rows, it is advisable to commit not too often but not after too many rows either. In the literature, a commit is advised to be performed after very 10 000 rows. This value depends on the size of the rollback segment, a parameter set by the DBA. The more rows are updated without making a commit, the more space Oracle will need to save the content in case you rollback. On the opposite, if a commit is done frequently, it negatively affects the performance. A commit implies a lot of work such as synchronizing all the caches of all the current sessions. To make sure that a commit will not be performed after 20,000 rows or more, inserts or updates in the CIC DB are forced if the cache contains 10,000 rows, even if it is not the last row. So every 10,000 rows the cache is reinitialized. The database currently used is maintained by the Central Database Support. They advised me to commit every 10,000 rows.

This value is set using `#define MAX_ROWS 10,000.` It is included in a header file "db_param.h", where all global variables related to the database are defined. So it is easy to update if there is any change in the rollback segment size.

7.2.1.5 Querying paths between 2 devices

The API of CIC_DB_lib includes functions which enable users to get detailed paths between two devices or a device and a device type which are part of a same subsystem.

The first idea was to compute these paths using PL/SQL. However generating a destination table containing the detailed paths between 2 devices (which both have one subsystem in common) using PL/SQL is not the best method for two reasons:

- Getting the paths between 2 devices is dynamic, in the sense that the name of two devices is not known in advance. Besides there are around 10 subsystem groups who can want to query paths between different types of devices. The **DESTINATION_TABLE** will contain too many rows and will be not easy to manage if we generate a destination table for each query "Get the paths between device A and device B". Moreover nothing can be predefined contrary to generating the destination tables of the DCHP servers or the TFC switch.
- There is a performance issue. Getting paths between devices will be used to test links and to configure parameter values. Tests have been done between an implementation in C and PL/SQL code. C code was much faster as it has better memory and loop management.

Therefore algorithms to get detailed paths between device A and device B or device A and a device type have been implemented in C.

The main steps are:

1. Load all the links which are part of the same subsystems as device A. It is sufficient to load the links which are part of the same subsystems as device A as device A and device B are in the same subsystem. In other words, a part of the **CONNECTIVITY** table is loaded in cache.
2. Load the microscopic links of all the devices which are in the same subsystem as device A. So a part of the **MICROSCOPIC_CONNECTIVITY** is loaded in cache. Depending on the level of the granularity this table can be empty.

3. Find all the paths starting from device A and ending at device B. A path cannot contain the same device twice and the pattern **intermediate node-host node-intermediate node** is rejected. It is the same as developed in PL/SQL package. Also there is a check that the combination (input, output) is valid. If data arriving at a given input can go out from the given output, then the path is kept, otherwise, it is rejected. It is done using a C function which uses the **MICROSCOPIC_CONNECTIVITY** table loaded in memory to verify if the given (input, output) can communicate. As a remark, if the **MICROSCOPIC_CONNECTIVITY** table is empty, the function is not called as the given (input, output) pairs are always valid. The main advantage of this check is it is independent of the board connectivity. The same algorithm to get the macroscopic paths is applied to check that the given combination (input, output) is valid.
4. Format and return the detailed paths if there are any.
5. If not, we reverse the query. It means applying the previous steps by inverting device A and device B so that the results are independent of the direction of the query. Indeed we do not assume that the user gives the device A and device B in the right order. For instance, if a given dataflow is from VELO_hybrid_09 to VELO_TELL1_12. The user may ask get the paths between VELO_TELL1_12 and VELO_hybrid_09. The step 3 will find no paths. The links stored in the **CONNECTIVITY** table need to be reverted to find the correct paths.

If whatever the query is, the subsystems are fixed, then there is no need to redo step 1. However, if new links have been added, a reload can be requested.

The query “get all paths through a device” is based on the same steps.

In any case, there is a timeout to avoid long computation of paths. It can happen if the user sets all the links to bidirectional and if there are a lot of devices with more than 100 ports.

7.2.1.6 Error Handling

There is error handling for the following errors:

- Unsuccessful malloc if a memory allocation fails;
- NO_ROWS_SELECTED, if the result of the SELECT statement is empty;
- BUFFER_TOO_SMALL, if the size of the buffer to which the result should be copied;
- COULD NOT UPDATE ALL THE ROWS with the name of the function which fails. This is the case when some of the rows have not been updated during a bulk update. It is the case if the where one condition is not satisfied such as mistypes. For instance,

  Update functional_devices set nodeused=1 where devicename = ’VELO_TLL1_23’. If VELO_TLL1_23 does not exist, this update will not work.
- COULD NOT INSERT ALL THE ROWS with the name of the function which fails. It means that some of the rows have not been inserted. This is the case when there is also a mistype. The device RICH1_TELL1_10 is of type RICH1_TELL1. When the user has inserted this device, he has attributed RICH1_TELL as device type, which does not exist, instead of RICH1_TELL1.

The Oracle errors explaining why a statement failed, such as a violation of a constraint or parent key not found are returned using the OCIError function. The complete Oracle error is returned in ErrMess, present in all functions of CIC_DB_lib. Besides the error which causes the failure, the name of function is also given.

7.2.1.7 Building CIC_DB_lib

The CIC_DB_lib contains 157 functions and 58544 lines. It consists of 4 header files and 27 source files. The library has been compiled with MCV.Net v7.1 on Windows and gcc version 3.2.3 20030502 (Red Hat Linux 3.2.3-56) on Linux.

All the functions are documented and available on the LHCb Online web site [5].

7.2.2 Features of CIC_DB_lib

The following features have been checked to comply with building autonomic tools.

7.2.2.1 Memory management

One of the common problems when implementing a library is to know which application should allocate the memory. To avoid any problem related to memory allocation, the application which uses CIC_DB_lib should allocate the memory. CIC_DB_lib will not allocate memory. However the application or the user will have to provide size of allocated buffers. For each function returning a list of devices, the user needs to specify the allocated size of the buffer in which the result should be put. If the size is too small, I put the required size in the variable which indicates the buffer size. If it is sufficient, I also put the real size needed here so that the application does not have to loop uselessly to extract the elements of int* or of char* as described in 7.2.1.4.

Example of use:

```
Int      GetHistoryOfFunctionalDevice(char*      functional_devicename,char*
functionaldevice_history, int & len_history, char* min_date, char* max_date,
char* ErrMess)
```

This function returns the history of a given device in `functionaldevice_history`. The application which calls this function has to put the allocated length of this parameter in `len_history`.

If the returned result can not be copied in `functionaldevice_history`, the size needed is put in `len_history` and I returned in `ErrMess`, BUFFER_TOO_SMALL. If the allocated size is enough, I copy the result in `functionaldevice_history` and specify the length in `len_history`.

7.2.2.2 Security

There was no security issue for the DB access as the CIC DB will be installed in a local network that is not accessible from the outside world. So there was no need for data encryption for instance. The only thing is that the user needs to connect the CIC DB by providing DB_name, login and password.

7.2.2.3 Consistency

7.2.2.3.1 Insert, update and delete information by block

Insert, delete and update functions have been implemented to ensure the consistency of the data in the CIC DB. The first thing is the block (a group of elements which needs to be updated or inserted or deleted within the same transaction) insertion (or update or deletion). It means that groups of data are inserted (or deleted or updated if required) together, using the same function. All these queries are part of the same transaction. Using this method, it avoids the user from calling several functions and it also prevents from forgetting to insert or update data.

Consider the following scenario. A user wants to insert a functional device. According to the table schema, inserting a functional device implies an insert in the **HARDWARE_DEVICES**, in the **FUNCTIONAL_DEVICES** and in the **DEVICE_HISTORY** tables. To allow efficient data consistency, the 3 inserts are done within the same SQL query. It consists of three consecutive SQL inserts. The main advantage is that if one of the inserts fails, the whole inserts fails. This is because of the use of foreign key constraints. Indeed, the first statement inserts into **HARDWARE_DEVICES.** Then the second inserts in **FUNCTIONAL_DEVICES** only if the first insert is successful. Finally the third ones inserts in **DEVICE_HISTORY** which can be done only if both previous inserts are successful.

The input parameters of **InsertFunctionalDevices** include all the table columns of **HARDWARE_DEVICES** and **FUNCTIONAL_DEVICES.**

Inserting by block is also used for port. If it is a DAQ device (i.e. it has an IP address), information will to be inserted in **FUNCTIONAL_PORT_PROPERTIES**, **HARDWARE_PORT_PROPERTIES** and **IPINFO** tables within the same SQL block.

The same concept has been applied for the UPDATE.

In the API, there are functions to delete links, ports, devices and device types. The user cannot delete a device if he has not deleted the ports of the device before.

When an insert, an update or delete affect the TFC or DAQ connectivity, a PL/SQL function part of **routingtable_pck** package updates the content of **DESTINATIONTABLE, ROUTINGTABLE** and **PATH_LINES** tables.

7.2.2.3.2 Use of status diagrams for inventory

Inventory information requires a lot of checking to avoid inconsistency. To update the status of a device, there are three functions (and three for updating board component status):

```
ReplaceFunctionalDevice(char*devicename,char*
new_device_status,char*user_comments,char*
status_datechange,char*serialnb_replacement,char*replace_date,  char*
ErrMess)
```

which allows the user to replace a functional device IN_USE by another hardware device (serialnb_replacement). The status of the hardware device which has been replaced must be specified (new_device_status). If the user sets serialnb_replacement to “none”, it means that the functional_device has status “NONE”.

SetToTestUseStatus(char* devicename, char* user_comments, char* status_datechange, char*serialnb_replacement, char* testboard_name, char*replace_date, char* ErrMess)

which allows replacing a functional device by another hardware device. The status of the hardware which was occupying the functional device is set to "TEST" and occupies a test board (testboard_name).

With these two functions any hardware device with status "IN_USE" can go to another status (TEST, EXT_TEST, DESTROYED or IN_REPAIR, SPARE).

UpdateHWDeviceStatus(char*devicename,char*new_device_status,char*new_location,char*user_comments,char* status_datechange,char*functional_devicename,char*ErrMess)

allows the user to set the status of a hardware device which does not have the status "IN_USE" to another status {EXT_TEST, DESTROYED or IN_REPAIR, SPARE, IN_USE}.

Using the following diagrams (see Figure 90, Figure 91, Figure 92, Figure 93, Figure 94) all the required checks can be made when updating the status of a device. Everything written in orange means it is an input parameter (provided by the user).

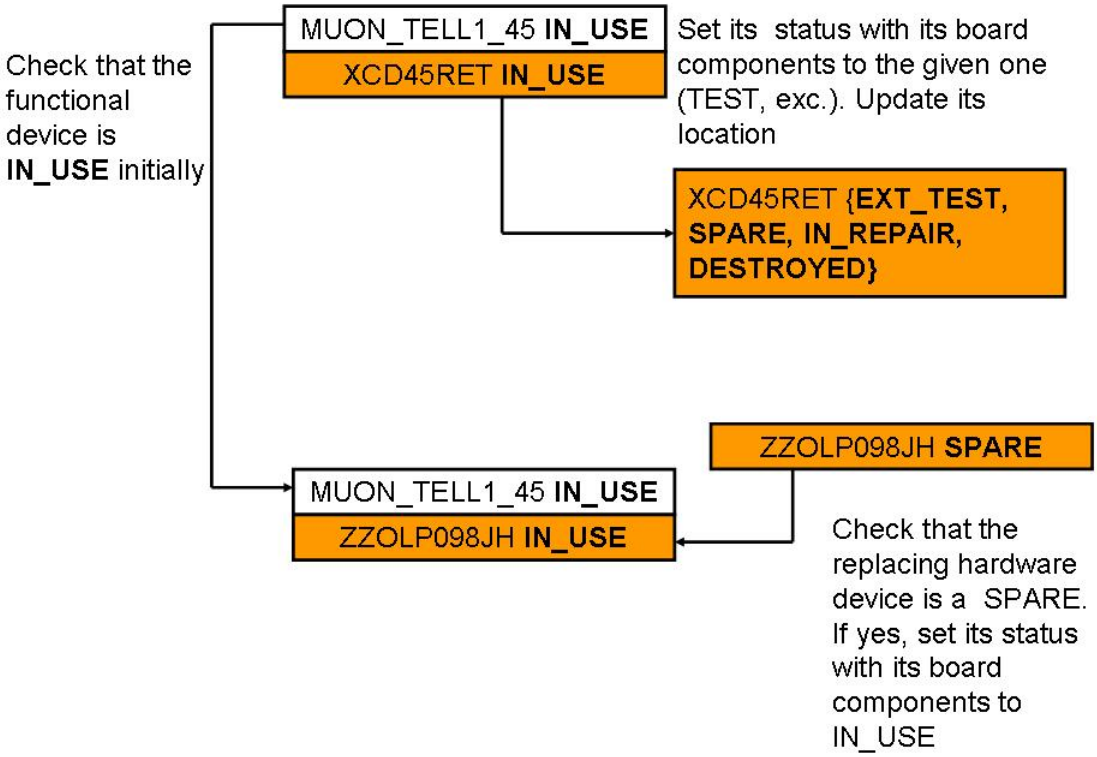


Figure 90. Replacing a hardware device.

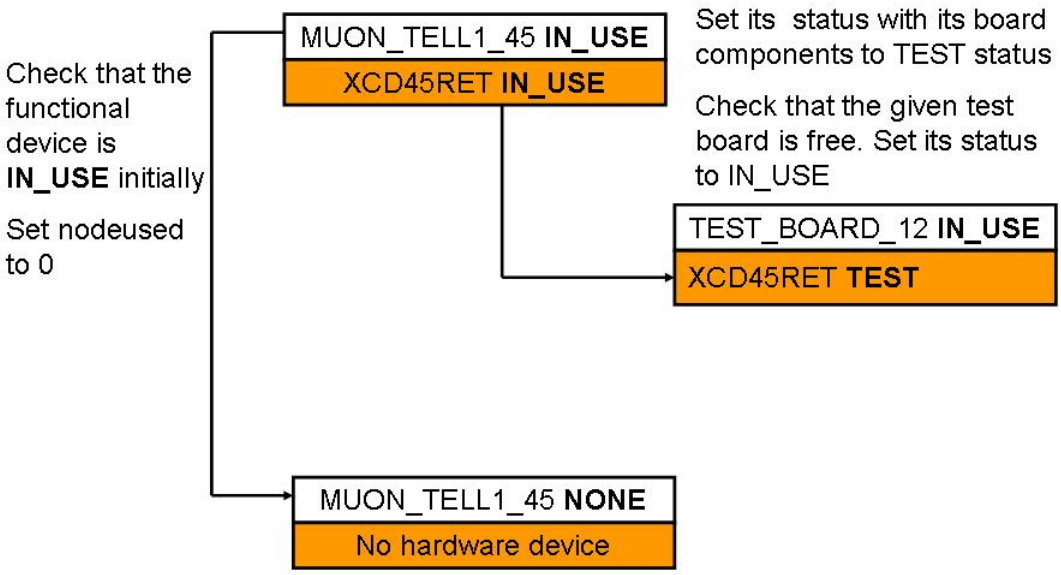


Figure 91. Setting the status of a hardware device to "IN_USE", with no replacement.

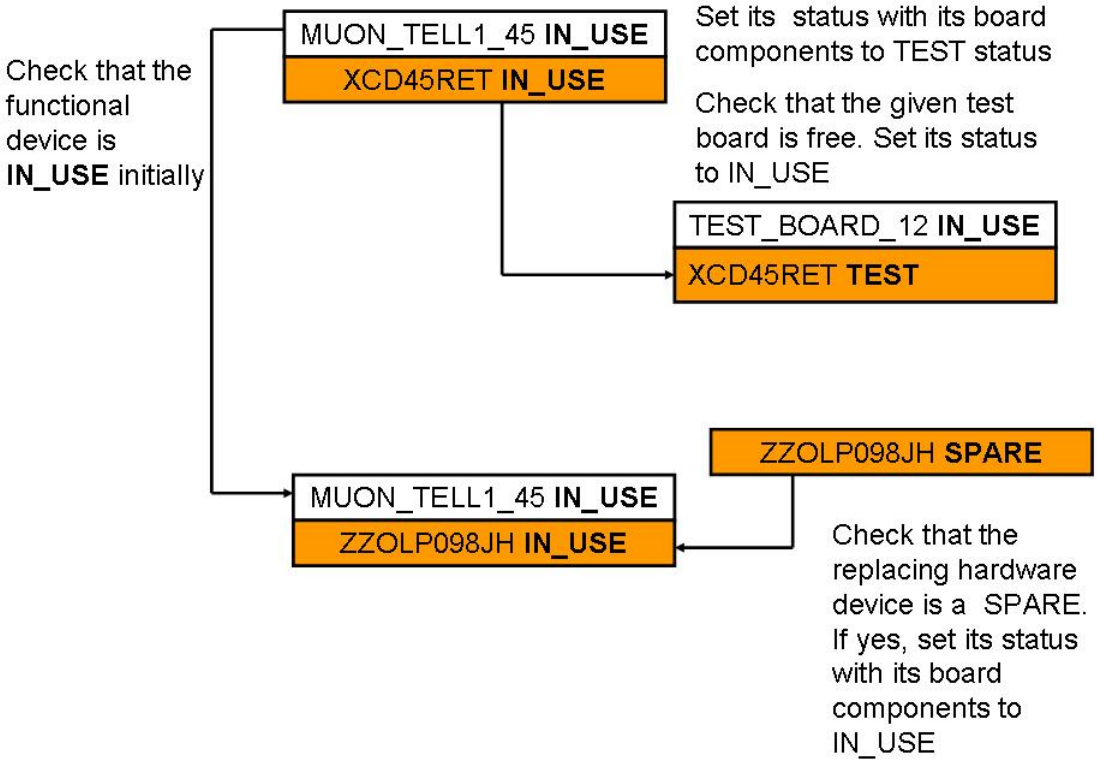


Figure 92. Changing the status of a hardware device from "IN_USE" to "TEST", with replacement.

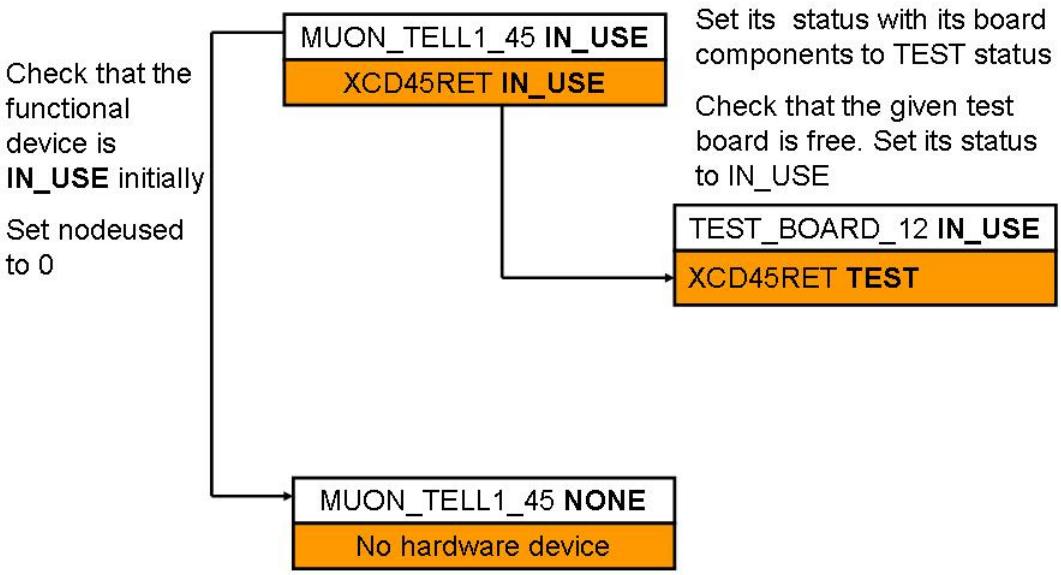


Figure 93. Changing the status of a hardware device from "IN_USE" to "TEST", with no replacement.

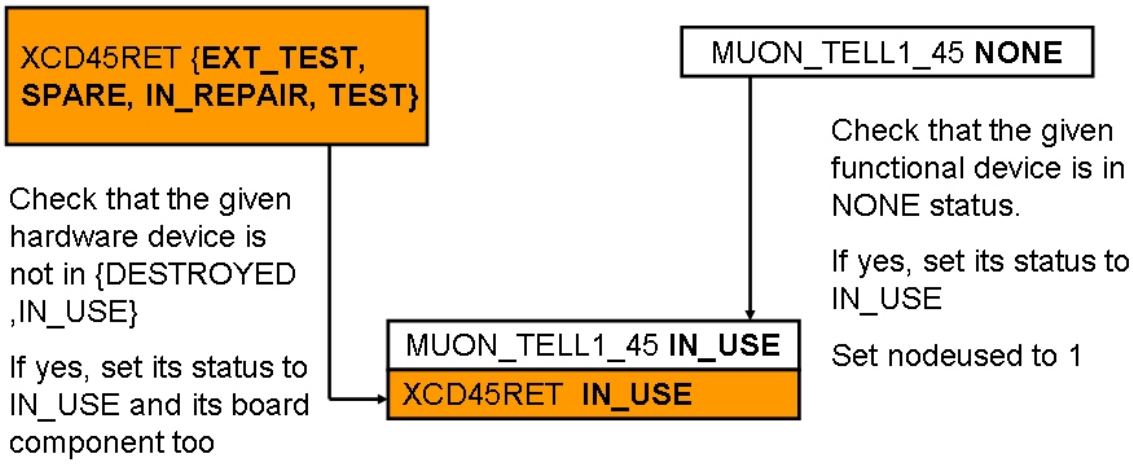


Figure 94. Updating the status of a hardware device to "IN_USE".

Each status change is inserted into the **DEVICE_HISTORY** table by calling functions from the CIC_DB_lib.

7.2.2.4 Concurrency

Functions including insert or update statements have been tested in multi-user environment. In other words, different processes have executed the same function

with different input parameters. This check was made to ensure that there was no blockage.

The tests have been performed on Windows using a C application which is based on the **CreateProcess** function. This function allows executing a function with specific command lines.

In fact, the different insert or update statements which correspond to a transaction were executed sequentially. During the tests, the “commit” or “rollback” statements were essential as it they prevent a SQL statement from blocking.

Through these tests, I could notice a problem when an Oracle sequence is created dynamically (in the **routingtable_pck**) and then it is rewound to 1 for the next call. The problem was that one of the functions was using the sequence and the other one wanted to rewind. Then the two functions raise an Oracle error. The problem is that Oracle sequences are the same and common to all the sessions unlike temporary tables which are bound to a session.

7.2.2.5 Autonomics

CIC_DB_lib has been implemented so that the tools which use it can reduce the human intervention and be self-adaptive in case of changes in the connectivity or the inventory.

This has been achieved by:

- understanding the system (the architecture, the dataflow, what is allowed what is not);
- anticipating the different failures or changes which can happen (devices or links out of order, swapping two devices, etc.);
- providing the maximum of flexibility (possibility to test a part of a system by disabling/enabling some links or some devices, etc.).

7.2.3 Issues

The difficulties which occur during the implementation of the library were the following:

- Define a complete API using the use cases. As the subsystems were built not at the same time, not all the use cases have been given at the same time. So the functions also have not been implemented at the same time.
- Best way to send back the results of a retrieval query, especially regarding paths so that the user or the application can easily extract and use the information required.

7.3 Bindings

7.3.1 Implementation of the PVSS CIC_DB_lib

A PVSS extension of the CIC_DB_lib has been implemented to permit interactions with the CIC DB from PVSS.

It has been implemented using GEH [6]. For each function of the CIC_DB_lib, a wrapper has been written based on TextVar and IntegarVar C++ classes. There is one

source file which contains all the wrappers. It has 10770 lines. The PVSS interface has been included in Appendix H.

The code below shows an example of the PVSS wrapper for the function DBConnexion. It is a function used to connect to the CIC DB.

```
DBConnexion(char* dbname,char* login,char* passwd,char* ErrorMess) :

/*****************************************************************
************/
/**
       * Connect to the database taking 3 arguments and returning an
integer value.
       * @param server : name of the database.
       * @param usr : user login.
       * @param pwd : user password.
         * @param errMess : return the ErrorMessage in case of
failure (otherwise NO_ERRORS in + fct name)
       * @return 0 if the disconnection is successful
       */
/*********************************************/

IntegerVar*  PVSSDBConnexion(TextVar*  server,TextVar*  usr,TextVar*
pwd,TextVar* errMess)
{
static IntegerVar c=0;
std::string dbname = server->getValue();
std::string login = usr->getValue();
std::string passwd = pwd->getValue();
int len_buffer=600;
char* ErrorMess=new char[len_buffer];

c=DBConnexion((char*)dbname.c_str(),(char*)     login.c_str(),(char*)
passwd.c_str(),ErrorMess);
errMess->setValue(ErrorMess);
delete [] ErrorMess;
return &c;
}
```

The extension exists both on Linux and Windows.

However I have encountered a compiler version problem. On Windows, Microsoft Visual 6.0 has to be used to compile it as the GEH has been built using this version.

Then on Linux, I have to use the compiler gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-113), an old version as the GEH has been compiled with this version. It was also not easy to get such a machine with this compiler version.

All the functions are documented and available on the LHCb Online web site [5].

7.3.2 Implementation of the Python CIC_DB_lib

There is also a Python binding. It is used by CDBVis, which will be explained in the next chapter. To interface it in Python, I used Boost [7]. This is an open source project which allows any bindings from C++ to Python. It is very efficient especially to manage and convert C pointers into Python objects. Boost has been integrated in Gaudi which is an LHCb Computing framework. The main advantage to use Boost integrated in Gaudi is the maintenance of the module and the integration in the LHCb environment.

Here also I had to write a wrapper for all the functions of the CIC_DB_lib.

The result is a module (lib) cicDBpython which can be used in Linux and Windows. In a python script, the library can be used by doing "import (lib)cicDBpython".

There was a need to implement two classes CICDB and CICDBEXCEPTION, to provide convenient error handling.

The attributes of CICDB are the connection parameters i.e. DB name, login and password. The methods of CICDB correspond to the functions implemented in the CIC_DB_lib. The CICDB class is presented in Appendix I.

The code below shows an example of the wrapper for the `DBConnexion(char* dbname,char* login,char* passwd,char* ErrorMess)`. It is the function used to connect to the CIC DB.

```
int CICDB::PyDBConnexion()
{
int c=0;
int len_buffer=1000;
char* ErrorMess=new char[len_buffer];
string ErrorMess_copy;


c=DBConnexion((char*)_dbname.c_str(),(char*)_login.c_str(),(char*)_pa
sswd.c_str(),ErrorMess);
 if(c!=0)
 {
       ErrorMess_copy=ErrorMess;
       delete [] ErrorMess;
       throw CONFDBEXCEPTION(ErrorMess_copy);
 }

 delete [] ErrorMess;
return c;
}
```

Here I had two main problems:

- Handling of the in-out parameters[13] as it does not exist in Python. Some of the results have been concatenated. For instance, the C function

  ```
  int GetAllPathsPerDevice ( char * systemnameList, char * devname, int
  & len_array, int * lkid_list, int * pathid_list, int * link_pos_list,
  int reload_connectivity, int delete_connectivity, char * ErrMessage )
  ```

  becomes in Python,

  ```
  vector< string > CICDB::PyGetAllPathsPerDevice ( string systemname,
  string devicename, int reload_connectivity, int delete_connectivity ).
  ```

  The return value of this function is a vector of strings. Each row of this vector corresponds to the concatenation of pathid_list| link_pos_list|lkid_list|.

- Compiler and Python version. It requires MCVNet v7.1 on Windows. It did not work with the previous version of Microsoft Visual because of Boost. Moreover the module cicDBpython can work only with Python 2.3.4 because of the Boost module.

Finally, all the functions are documented and available on the LHCb Online web site [5].

[13] By in-out parameters, we mean an input parameter whose value is changed by the function and is returned. It is both input and output parameters.

7.4 A PVSS library for recipes

The CERN PVSS Support has implemented a PVSS library to load and save recipes into/from PVSS. The documentation about the functions is included in the framework [8].

This library provides functions to load and save a recipe including versioning mechanism. It also includes functions to get the list of devices in a hierarchy, to get the list of available recipes (a device can be specified)

7.5 Conclusion

In this chapter, the object layer has been described. Two Perl scripts ("dhcpCfg_generate.pl" and "dns_generate.pl") have been used to generate the config file for both the DHCP and DNS servers. The "dhcpCfg_generate.pl" script which creates the DHCP config file embeds PL/SQL function from the **routingtable_pck** to build the destination table.

CIC_DB_lib with its two bindings (PVSS and Python) enables non-DB expert to query, update and insert information about connectivity and inventory. Compiler version problems were the main issue when building these two extensions.

CIC_DB_lib has been built to permit to implement tools (which use it) to be adaptive and smart, as any autonomic tools by reducing human intervention. To get and save information about recipes, the CERN PVSS Support has implemented a PVSS library which can be used in PVSS panels.

Implementing autonomics tools is made possible by using the libraries (CIC_DB_lib ant its two bindings), PVSS and its frameworks. Thus it will permit to develop a single-click control system. However there is still the need to start and stop the experiment, which can be done only by a human being.

The next chapter explains how these libraries have been integrated in the GUI layer.

References

[1] L.Abadie, Generating the DHCP config file using confDB, LHCb Internal Note, June 2006. LHCb-2006-038.

[2] ORACLE, Oracle® Database, SQL Reference, 10g Release 2, December 2005. B14200-02. 1428 p.

[3] ORACLE, Oracle Call Interface, Programmer's Guide, 10g release 2. ORACLE PRESS, OSBORNE. B14250-02, November, 2005. 1258 p.

[4] ORACLE. OCCI (Oracle C++ Call Interface) Programmer's Guide, 10g Release 2 (10.2). ORACLE PRESS, OSBORNE, December 2005, B14294-02, 474 p.

[5] CIC_DB_lib Documentation and its two extensions, http://lhcb-online.web.cern.ch/lhcb-online/configurationdb/download_libraries.htm

[6] GEH, Generic External Handler, http://itcobe.web.cern.ch/itcobe/Projects/Framework/Download/Components/GenericExternalHandler/welcome.html

[7] Boost Python, http://www.boost.org/libs/python/doc/

[8] PVSS library for recipes, http://itcobe.web.cern.ch/itcobe/Projects/Framework/Download/welcome.html

Chapter 8 Structure of the GUI layer

This chapter describes the different GUIs which have been built on top of the object layer. First we explain the features and the implementation principles of CDBVis, a graphical editor to navigate through the CIC DB. Secondly, we explain how the XML config files generated by the two Perl scripts are converted using XSLT to config files which are accepted by the DHCP and DNS servers. Finally we present some PVSS panels which have been implemented to configure modules and handle partitioning. We also show that both information about configuration parameters and connectivity are combined by the ECS so that a single-click system can be built.

8.1 CDBVis, a graphical editor

8.1.1 Features

CDBVis [1] enables a non-DB expert user to view and insert connectivity information.

Its features are the following ones:

- Creation of device types, devices, ports and link types and links per subsystem, mass insertion has been developed;
- Update device types, devices, ports and link types and links per subsystem;
- Implementation of a cache which saves everything created by the user. The cache is emptied when the user confirms that data should be sent to the CIC DB;
- View the list of device types, devices, ports per subsystem;
- View the neighborhood connectivity of a device;
- View all the paths which go through a selected device;
- Common features related to graphical tools such as zooming, structure duplication, status bar progress, undo and redo etc.

This tool is very useful to check that the connectivity of subsystem was correctly inserted. Figure 95 to Figure 101 show snapshots of CDBvis:

- Figure 95 shows the first panel when the tool is started. It is composed of 3 parts, a Selection Window which displays a tree with 4 levels (subsystems, device types, devices and ports), a Visual Window which displays the connectivity between devices and an Information Window which gives some information when an element of the tree is selected.
- Figure 96 shows on the left, an example (not complete) of MUON hierarchy which is displayed in the Selection Window. It contains two device types FEE and M5R4. There are 5 devices of type FEE. (FEE_MUON_1 is one device). On the right, there is the Information Window which displays information about FEE_MUON_1. The attributes correspond to the attributes of the DEVICE entity as explained in Chapter 5.
- Figure 97 shows how objects can be created. On the top right, there is a panel to create device types. The user has to fill in the different attributes of the entity DEVICE_TYPE such as the name, the number of input and output ports, a description, etc. On the top left, there is a panel to create devices. Creating

multiples devices in one go is possible but updating afterwards the serial number. The user has to provide the name, the device type, the serial number, the responsible, etc. When the user creates a device, it covers both the functional device and the hardware device. It avoids doing two different insertions. On the bottom left, there is a panel which allows creating a port. A device cannot be inserted if the user does not save any ports. Indeed a device without ports does not make sense as it means that it cannot be connected! Here also the parameters to fill in correspond to the attributes of the entities PORT_PROPERTIES and HWPORT_PROPERTIES and IPINFO. By doing such information grouping, we avoid having many panels and steps for the users. Also it is easier to detect errors. For instance, let us assume that the user first had to create 10 hardware ports. Then it has to create the 10 associated functional ports. Assume that one of the functional ports had not been created properly or vice versa; the hardware port associated should be removed too. Performing a rollback will delete the ten rows for both hardware and functional ports provided that there were no commit between the two distinct insertions. So one has to reinsert the 10 hardware ports and the 10 functional ports. It is not very convenient for the user. In our case, the insert has been implementing in such way that either both or none of the ports (hardware, functional) are inserted. Thus one row out of 10 could not be inserted properly, the user will just have to insert the row which fails (CDBVis gives the possibility to get which row to be inserted has failed). On the bottom right, there are two panels, one for creating the link types (for entity LINK_TYPES) and another one to create the link between devices (for entity CONNECTIVITY).

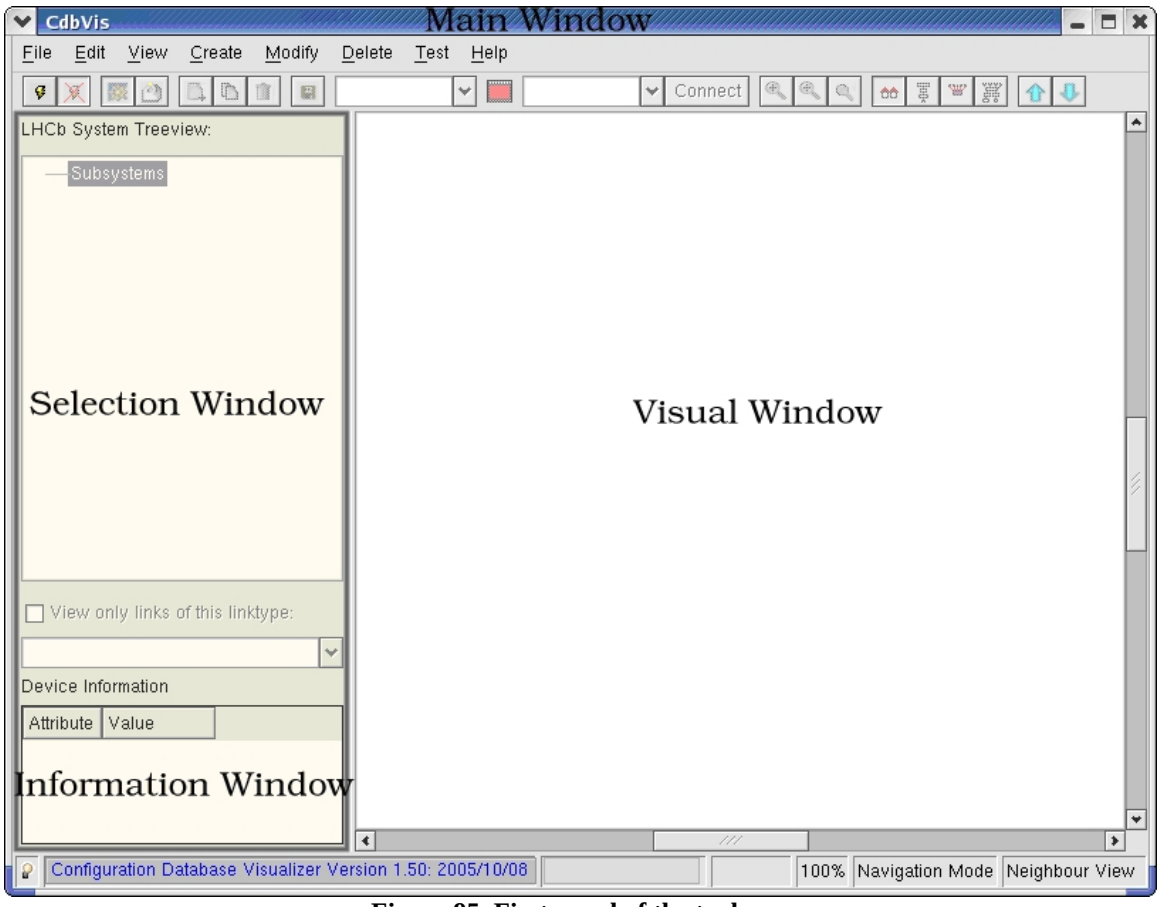


Figure 95. First panel of the tool.

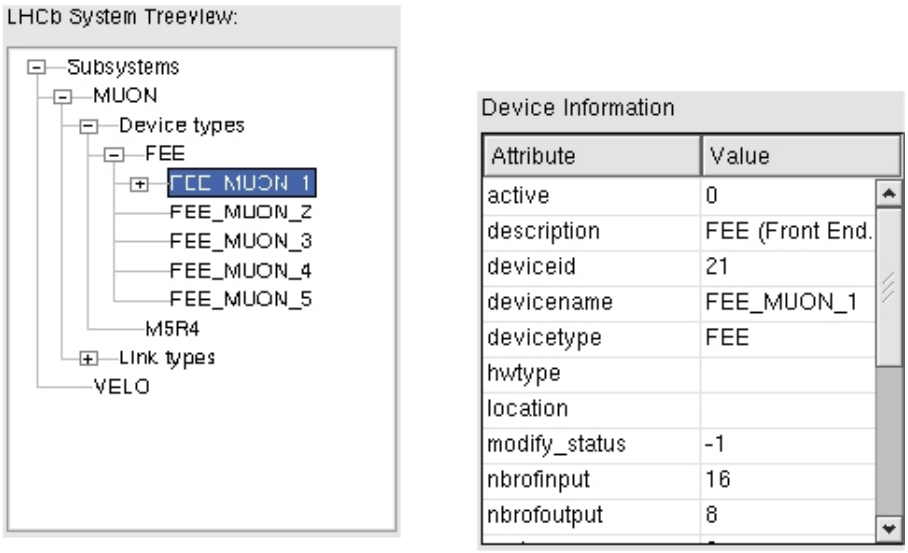


Figure 96. Browsing information via a hierarchy (left). Information about the device (right).

- Figure 98 shows the input and output connectivity of the TFC switch, called ThorV1_00. It is because we are in neighborhood view. With this mode, when the user clicks on a device in the Selection Window, the devices connected on its input and on its output are displayed. On its input it is connected to 16 readout supervisors (named OdinVX_YY, where X and Y represent numbers). On its output, it is connected to 15 TTCtx (from TTCtx_00 to TTCtx_14). By clicking on one of the link, information about this link is displayed on the Information Window.
- Figure 99 shows the input and output connectivity of the distribution switch in the DAQ system, called DAQ_SWITCH_14. On its input it is connected to the Force Ten, called DAQ_ROUTER_01 (via 10 links). On its output, it is connected to 40 farm nodes and to another distribution switch (DAQ_SWITCH_60) for the storage.
- Figure 100 and Figure 101 show two examples in the Path mode view. With this mode, when the user clicks on a device in the Selection Window, all the paths which go through this device are returned as pop up panel which lists all the paths found. Then the user selects one of the paths and this latter is displayed.

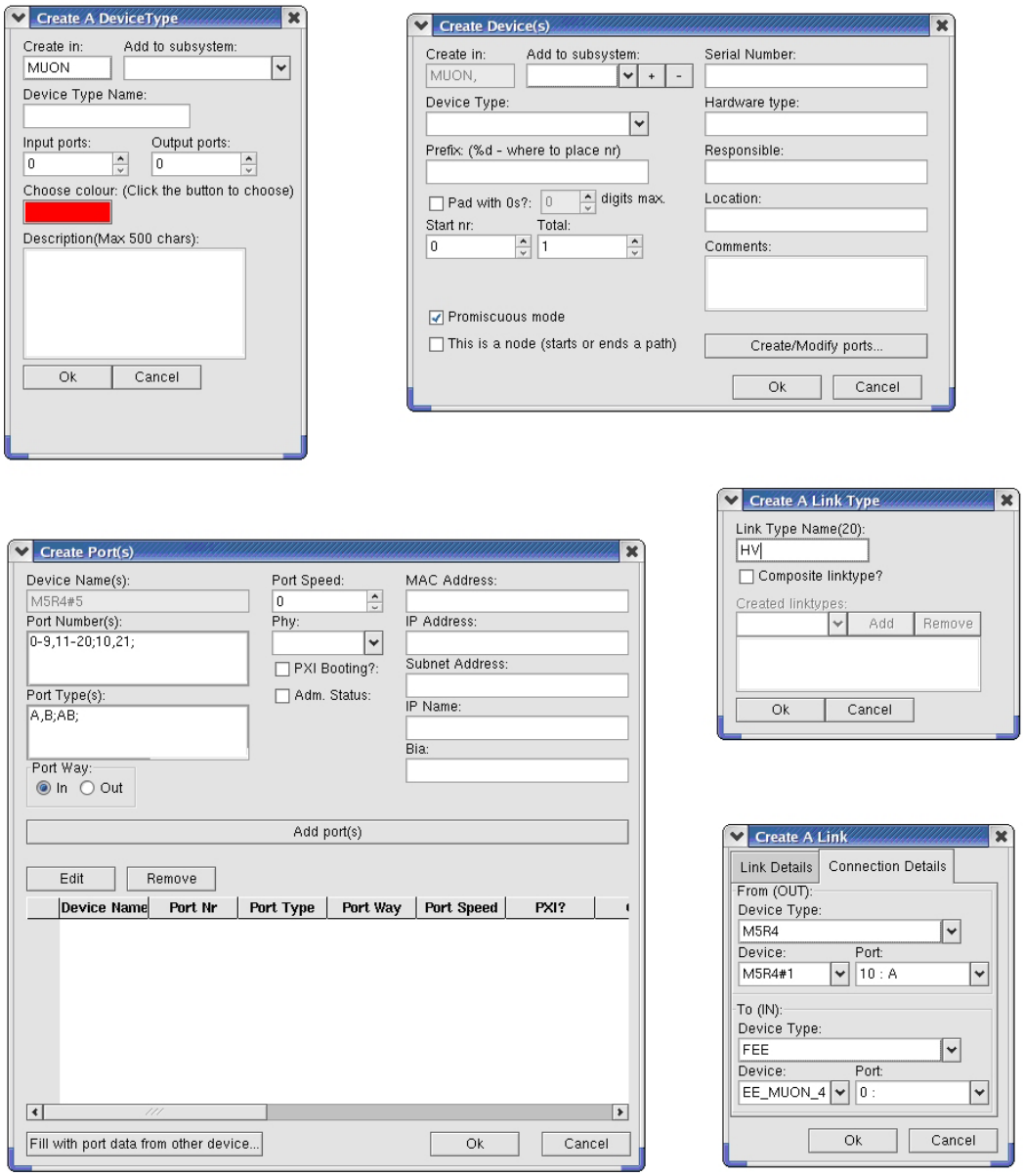


Figure 97. Creating objects. On the top left, a panel to create device types and on the top right, a panel to create devices. On the bottom left, a panel to create ports and on the bottom right a panel to create link types and a panel to create links.

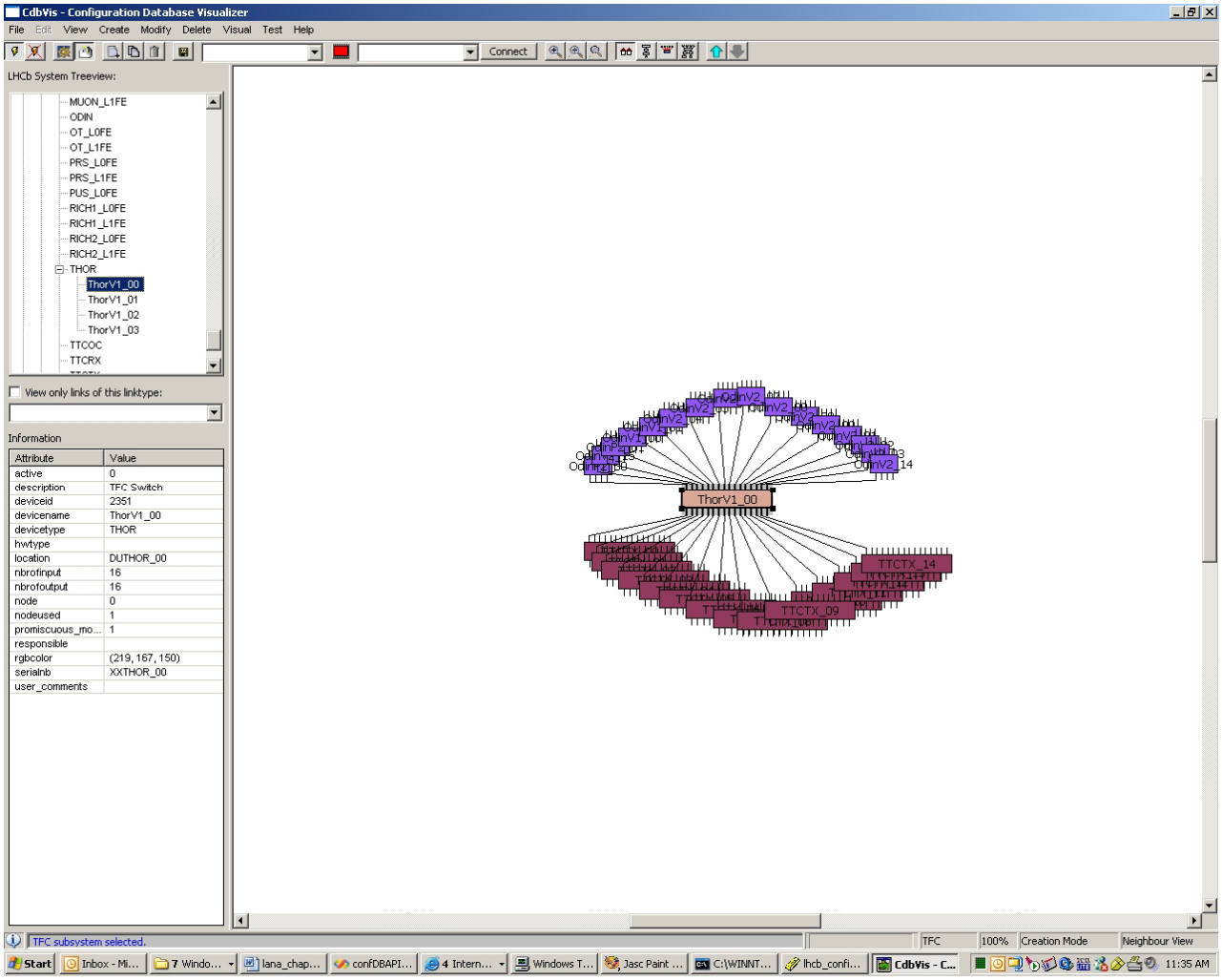


Figure 98. Neighbored connectivity for the TFC switch (ThorV1_00).

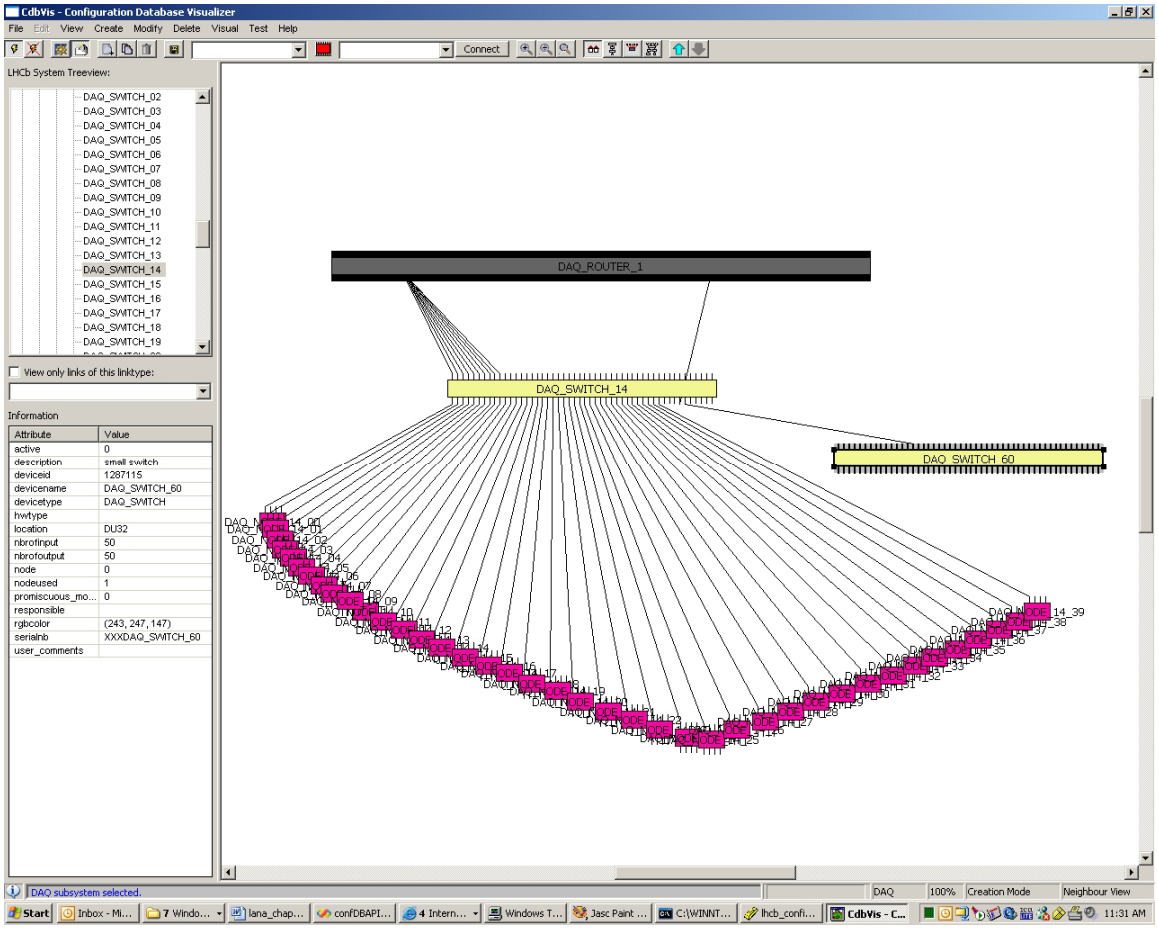


Figure 99. Neighbored connectivity for the switch DAQ_SWITCH_14.

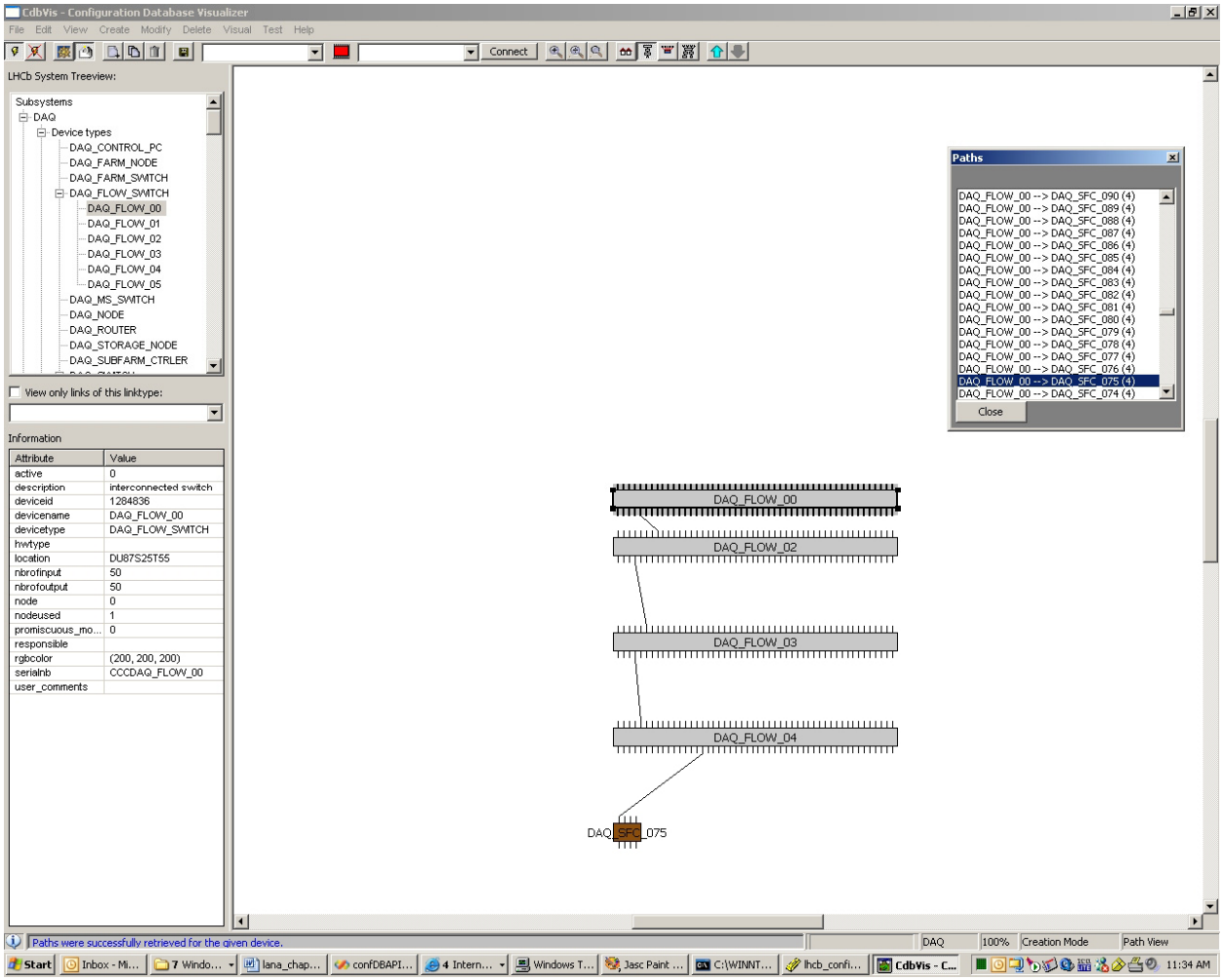


Figure 100. Path mode view: get paths through DAQ_FLOW_00.

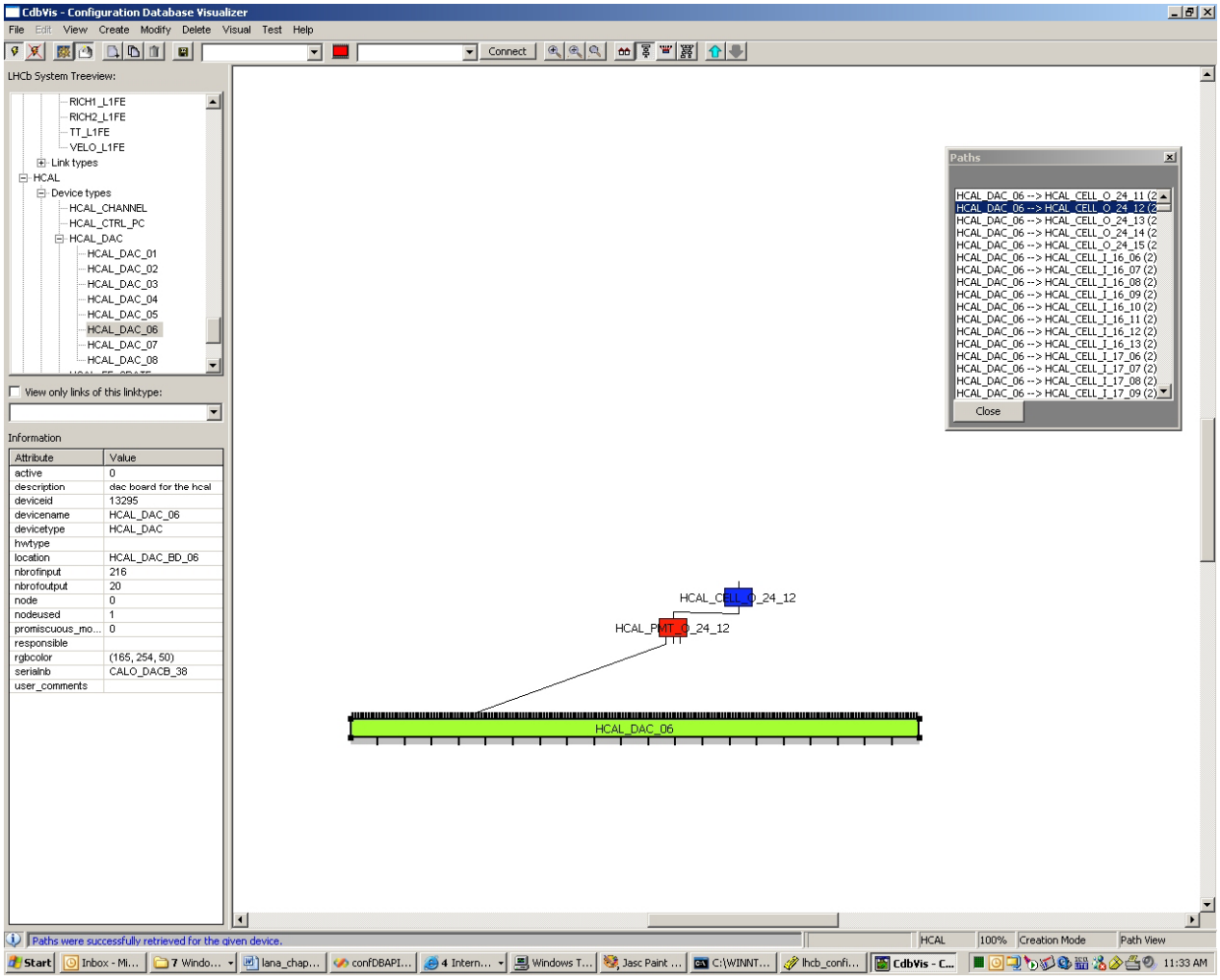


Figure 101. A Path going through HCAL_DAC_06.

8.1.2 Implementation

CDBVis is a graphical tool written in Python. Python has been chosen because it is commonly used in the LHCb software environment. It is an easy and convenient language to build a prototype. Also as Python is portable, the tool works both on Windows and Linux.

The graphical tool has been developed by a collaborative effort between many people, including myself. It uses two python modules:

- wxPython , a widget library to build graphical panels;
- (lib) cicDBpython (the Python binding of the CIC_DB_lib, see Chapter 7 section 7.3.2) to access and interact with the CIC DB (connection/disconnection, queries).

It has been programmed using objects and methods as shown in Figure 102 [1]. The classes correspond to the tables built for the connectivity and inventory/history schema.

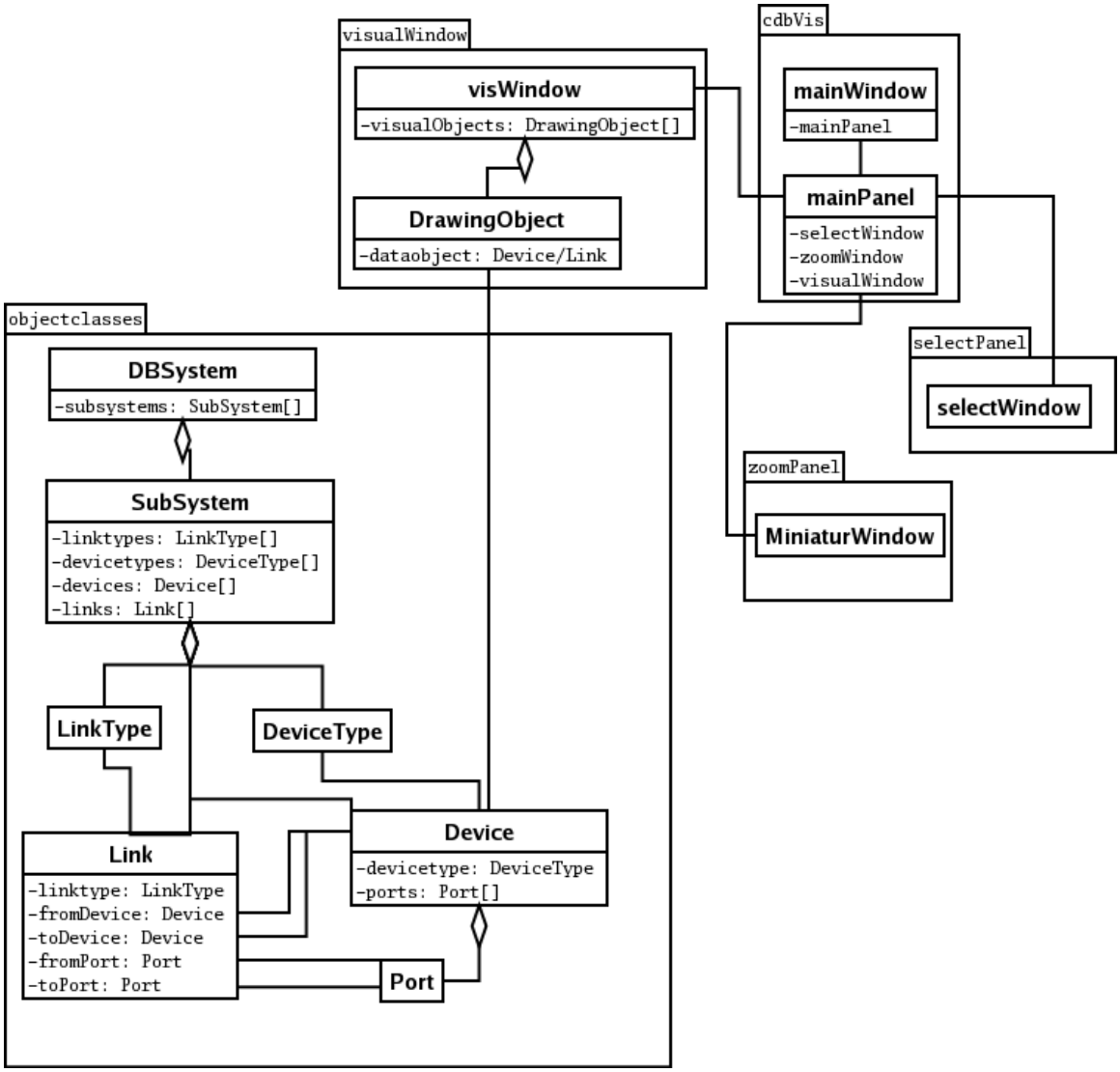


Figure 102. The class model above shows the relations between the classes in the different modules (shown as packages), and the member variables that are responsible for the association/reference between the classes are shown.

8.1.3 Issues

CDBVis permits the user to navigate through the connectivity of a subsystem stored in the CIC DB. The main problem we face when implementing CDBVis was to display the connectivity. There are different types of connectivity with different devices. For instance, the Force Ten router with its thousands of ports has to be readjusted compared to the other smaller switches.

Getting all the paths is quite time-consuming. It can go from a few seconds to a few minutes depending on the total number of paths found. For instance, Figure 101 shows paths for the device HCAL_DAC_06. It takes two minutes to find the 13952 paths and getting information about all the different links in the paths and create all the Python objects needed to display them. 1 minute and half is needed to retrieve the paths and half a minute to create the object. Creating all the Python objects can take a lot of time especially for the DAQ. It takes around 1 minute to get the 5384 paths going through the DAQ_SWITCH_14 and around 4 minutes to create the objects. The main problem is to create the object device DAQ_ROUTER_1 and its 1200 ports!

Each port is an object. Presenting the paths found to the user especially when there are thousands of them was not obvious. Currently, the user has to select one among many paths (from ten to thousands as mentioned previously). Once a path selected, there is a need to draw the path but it takes one or two seconds as all the objects have been created.

CDBVis works for most features but is still under work. New features should be added such as the possibility to view the microscopic connectivity of a device and also to insert it.

8.2 The DHCP and DNS config files

8.2.1 XML output

The two Perl scripts “dns_generate.pl” and ““dhcpCfg_generate.pl” print the result of the database queries on XML files. The conversion of the XML files into DHCP and DNS config files is ensured by XSLT. The main advantage is that the XML files are independent of the DHCP and DNS specific syntax.

The tags used in the XML files are listed in Table 17.

XML Tag	Used in DNS	Used in DHCP
<page>	Yes	Yes
<option_list>	Yes	Yes
<SOA>	Yes	No
<expire>	Yes	No
<retry>	Yes	No
<refresh>	Yes	No
<ttl>	Yes	No
<TTL>	Yes	No
<option>	No	Yes
<ethernet_add>	No	Yes
<row>	Yes	No
<ipadd>	Yes	Yes
<ipname>	Yes	Yes
<function>	Yes	No
<filename>	No	Yes
<subnet>	No	Yes
<subnetID>	No	Yes
<subnet_mask>	No	Yes
<rowset>	Yes	Yes

Table 17. List of XML tags used in the DHCP and DNS config files.

The code below shows an example of the XML output for the DHCP config file:

```
<?xml version = '1.0'?>
<page>
<option_list><option>ddns-update-style ad-hoc
</option></option_list>
< option_list ><option>deny unknown-clients
</option></option_list>
< option_list ><option>use-host-decl-names on
</option></option_list>
```

```
< option_list ><option>always-reply-rfc1048 true
</option></option_list>
<option_list><option>domain-name-servers 137.138.16.5
</option></option_list>
<options><option>option routers 137.138.1.1 </option></options>
<subnet>
<subnetID> 137.30.102.0 </subnetID>
<subnet_mask> 255.255.255.0 </subnet_mask>
<rowset>
<row><ethernet_add>00:00:DD:19:52:15</ethernet_add><ip_add>137.30.102
.103</ip_add><ipname>DAQ_NODE_02_03</ipname><filename>farm_nodes_imag
es.nbi</filename></row>
<row><ethernet_add>00:00:DD:19:52:16</ethernet_add><ip_add>137.30.102
.104</ip_add><ipname>DAQ_NODE_02_04</ipname><filename>farm_nodes_imag
es.nbi</filename></row>
<row><ethernet_add>00:00:DD:19:52:17</ethernet_add><ip_add>137.30.102
.105</ip_add><ipname>DAQ_NODE_02_05</ipname><filename>farm_nodes_imag
es.nbi</filename></row>.
```

The code below shows an extract of the DNS reverse XML file:

```
<?xml version = '1.0'?>
<page>
<option_list>
<SOA>137.26.in-addr.arpa.     IN     SOA     DAQ_CTRLPC_10_01.ecs.lhcb.
root.localhost.</SOA>
<serial>200609051</serial>
<TTL>86400</TTL>
<refresh>3h</refresh>
<retry>3600</retry>
<expire>4w</expire>
<ttl>3600</ttl>
</option_list>
<rowset>
<row><ipadd>137.26.in-
addr.arpa.</ipadd><ipname>DAQ_CTRLPC_10_01.ecs.lhcb.</ipname><functio
n>NS</function></row>
<row><ipadd>05.100.60.137.</ipadd><ipname>DAQ_CTRLPC_60_01.ecs.lhcb.<
/ipname><function>NS</function></row>.
```

8.2.2 Conversion using XSLT

Using XML::XSLT Perl library, the XML files are parsed using an XSLT sheet. There is one for the DHCP and two for the DNS (one for the forwarding and one for the reversing).

The tags listed in Table 17 are used as block delimiters and some of them are renamed as mentioned in Table 18.

XML Tag	XSLT use
<page>	Delimiter
<option_list>	Delimiter
<SOA>	Delimiter
<expire>	;expire
<retry>	;retry
<refresh>	;refresh
<ttl>	;ttl
<TTL>	$TTL
<option>	Delimiter
<ethernet_add>	hardware ethernet
<row>	Delimiter
<ipadd>	fixed-address
<ipname>	Host
<function>	Delimiter
<filename>	Filename
<subnet>	Delimiter
<subnetID>	Subnet
<subnet_mask>	Netmask
<rowset>	Delimiter

Table 18. Processing of XML tags in the XSLT stylesheet.

The generated DHCP config file looks like as below:

```
dns-update-style ad-hoc
; deny unknown-clients
; use-host-decl-names on
; always-reply-rfc1048 true
; domain-name-servers 137.138.16.5
;  option  routers  137.138.1.1  ;  subnet    137.30.102.0    netmask
255.255.255.0  {  group {
   host DAQ_NODE_02_03{
   hardware ethernet 00:00:DD:19:52:15;fixed-address 137.30.102.103;
   filename "farm_nodes_images.nbi";}
   host DAQ_NODE_02_04{
   hardware ethernet 00:00:DD:19:52:16;fixed-address 137.30.102.104;
   filename "farm_nodes_images.nbi";}
   host DAQ_NODE_02_05{
   hardware ethernet 00:00:DD:19:52:17;fixed-address 137.30.102.105;
   filename "farm_nodes_images.nbi";}
```

The forwarding DNS file is like:

```
$TTL86400
ecs.lhcb. IN SOA DAQ_CTRLPC_10_01.ecs.lhcb. root.localhost. (
200609051;serial
3h;refresh
3600;retry
4w;expire
3600;ttl
)
                ecs.lhcb. IN  NS  DAQ_CTRLPC_10_01.ecs.lhcb.
                DAQ_CTRLPC_60_01  IN  NS  137.60.100.05
```

The reversing DNS file is like:

```
$TTL86400
```

```
137.60.in-addr.arpa.      IN      SOA      DAQ_CTRLPC_10_01.ecs.lhcb.
root.localhost.(
200609051  ;serial
3h  ;refresh
3600 ;retry
4w  ;expire
3600  ;ttl
)

                  137.60.in-addr.arpa.          IN                    NS
DAQ_CTRLPC_10_01.ecs.lhcb.          05.100.60.137.      IN          NS
DAQ_CTRLPC_60_01.ecs.lhcb.
```

8.3 Use of PVSS panels

8.3.1 Handling partitioning

To handle partitioning, the TFC switch must be programmed. It means that we need to determine to which output port a given subsystem is connected and to which input port of the TFC switch a given readout supervisor is connected.

The destination table of the TFC switch allows getting the output port given a subsystem name. Indeed using the destination table, we get all the reachable host nodes, which correspond to TELL1 boards and consequently we know the subsystem.

The destination table of the TFC switch is created and maintained in the CIC DB.

To program the TFC switch, a local TFC control system has been implemented in PVSS.

As the destination table is fixed, a PVSS script gets the output port per subsystem from the CIC DB. The loading is done using **PVSSGetPortPerSubsystem**, also part of the CIC_DB_lib.

The signature is as follows:

Int **PVSSGetPortPerSubsystem** (string dfrom,
string subsystem_name,
dyn_string &pfrom_list,
string & ErrMess)

Example of use:

```
PVSSGetPortPerSubsystem("THOR_00", "VELO_A", pfrom_list, errMess);
```

Then to get the connectivity between readout supervisors and the TFC switch, **PVSSGetLkToDevID** is used. This function allows obtaining all the linkids connected to the input of a given device. This one is used, as the readout supervisors are directly connected to the TFC switch.

Example of use:

```
Dyn_int linkid_list;
String lkid_row;
Int devid=PVSSGetDeviceID_devicename("THOR_00",errMess);
Linkid_list=PVSSGetLkToDevID(2351,errMess);
For (i=1;i<=dynlen(Linkid_list);i++)
lkid_row=PVSSGetMacroConnectivityRow_lkid(Linkid_list[i])
```

Connectivity information is stored in a datapoint so that it can be accessed during the run of the project.

Figure 103 shows the PVSS user interface to program the TFC switch. The first thing is to select the subsystems part of the partition. Then, after clicking on the **FIND ODIN** button, the connectivity between the selected subsystems and the TFC switch is displayed in Figure 104.

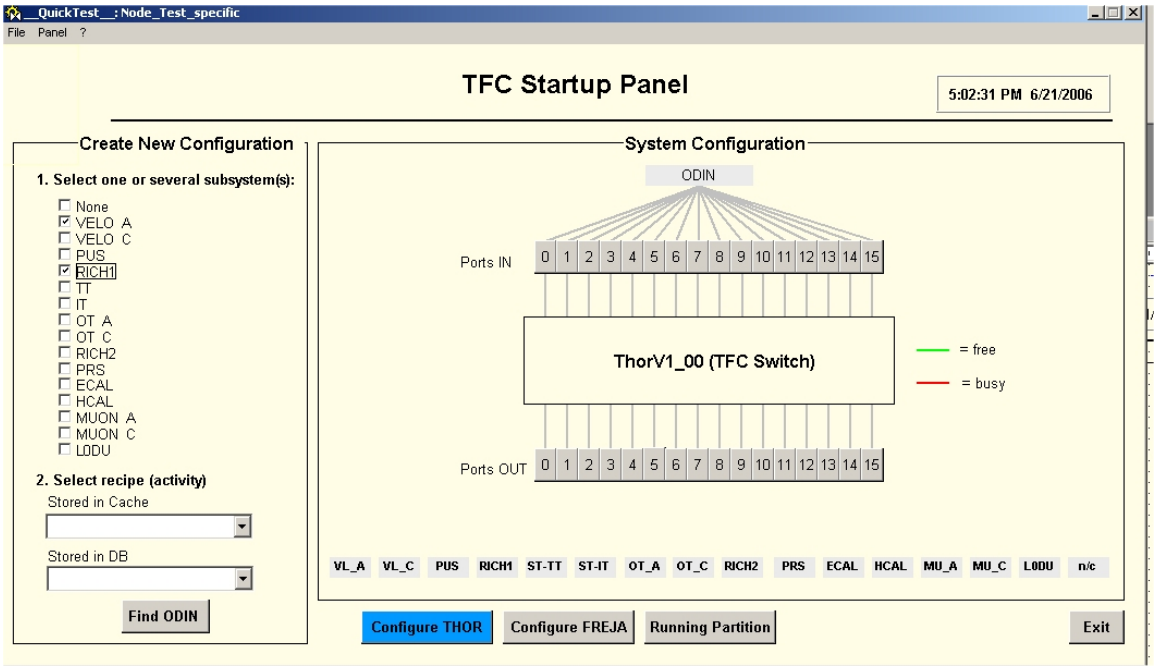


Figure 103. First step: select the subsystems.

The selected subsystems are no longer free as shown in Figure 104. Besides the connectivity between subsystems and the TFC switch, the first free readout supervisor is suggested and is connected to the TFC switch.

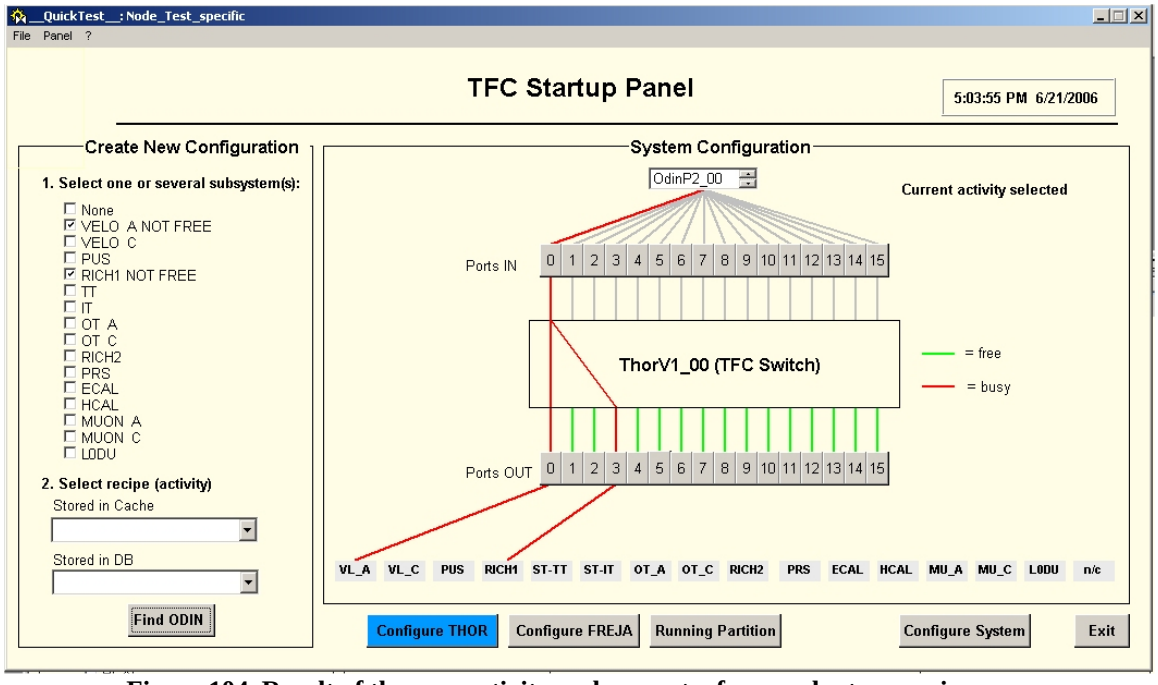


Figure 104. Result of the connectivity and suggest a free readout supervisor.

If the user is not satisfied with this one, he can select another one and the connectivity is automatically updated as shown in Figure 105.

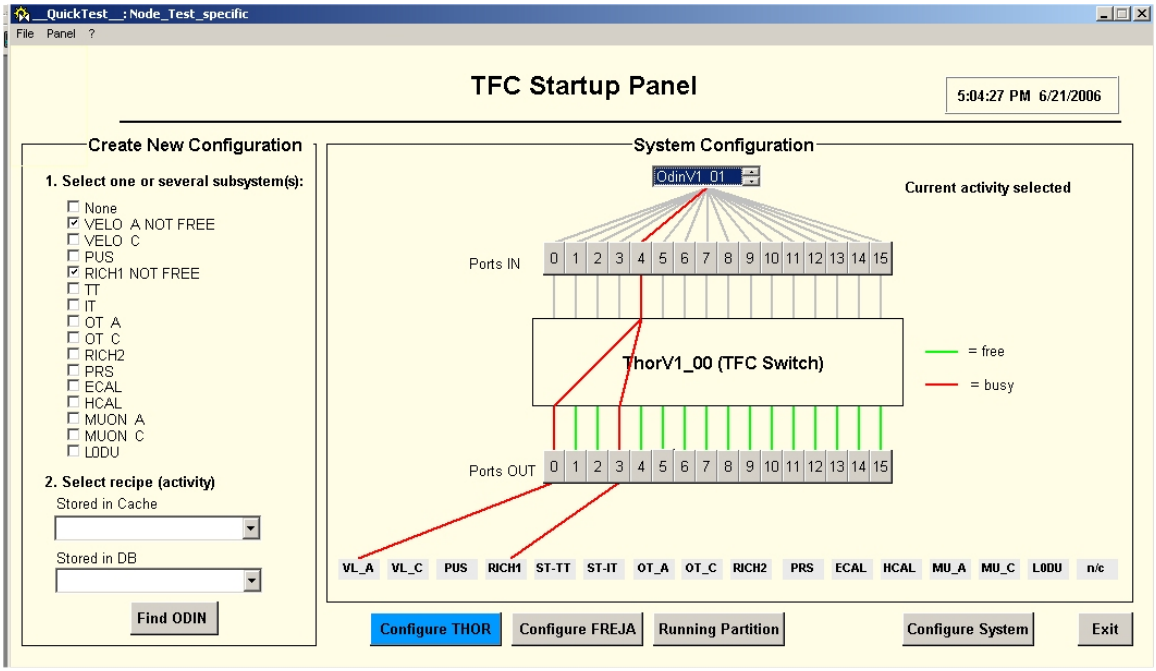


Figure 105. Select another readout supervisor, update the connectivity.

8.3.2 Recipes

The PVSS library for recipes has been integrated in LHCb specific components such as FwHW [2]. This is a PVSS tool which allows controlling hardware equipment with PVSS.

Once the hardware has been represented with data-points, the tool allows defining recipes and FSM states. It is also used to configure and monitor the hardware. Figure 106 shows the different parameters that can be configured. It consists of essentially registers (see address and sub address field), bus (I^2C, JTAG, LBUS, etc.) and FPGA code to download (see Figure 107). Using DIM, the commands Read, Write, Reset enable the user to interact with the hardware.

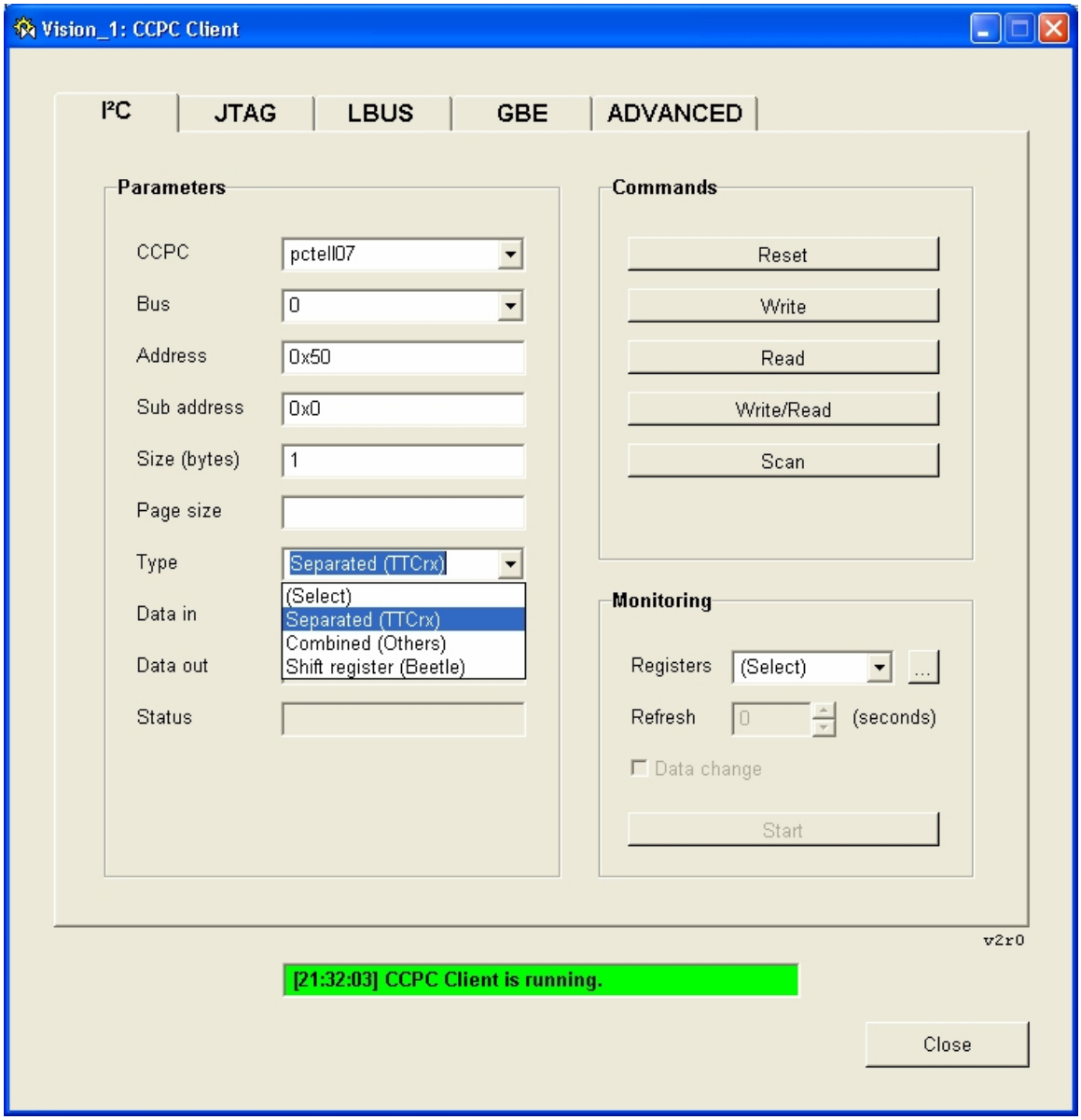


Figure 106. A PVSS panel to configure buses of a TELL1 board.

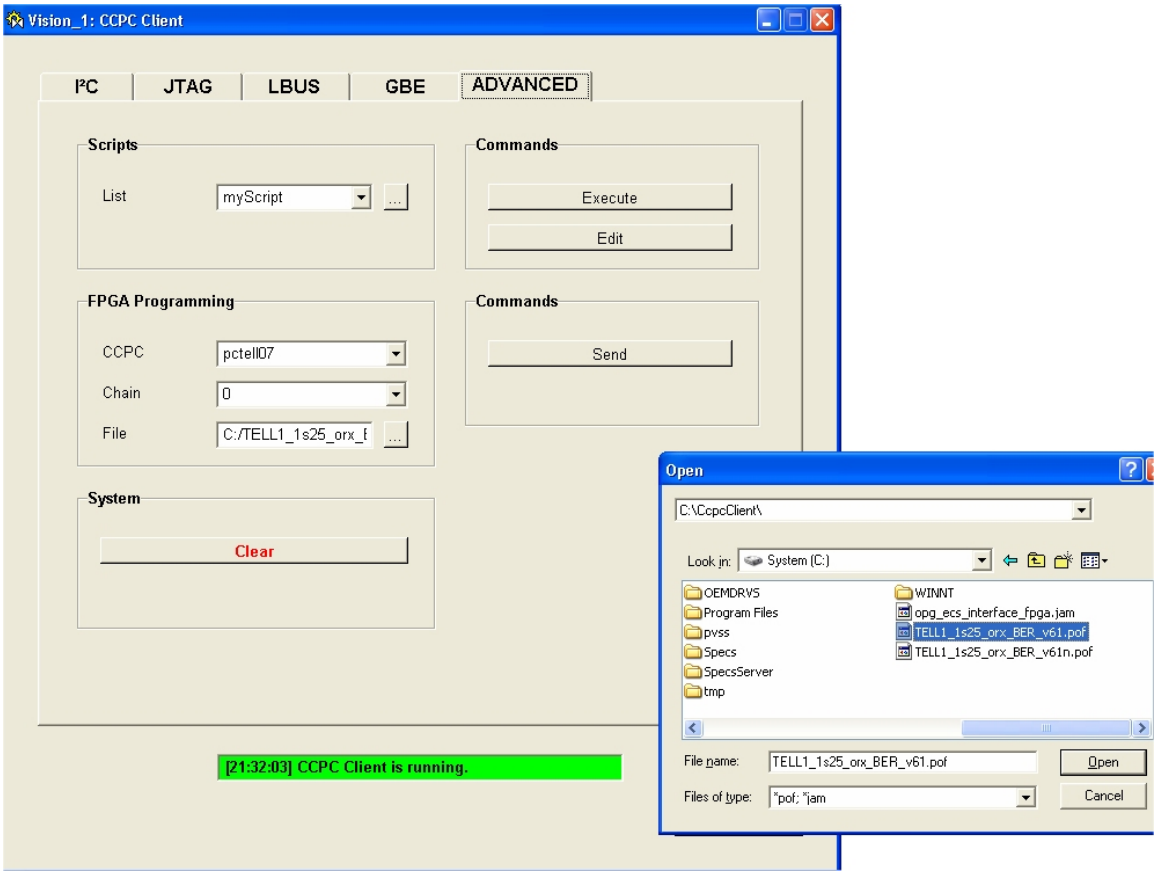


Figure 107. Downloading a FPGA code.

This library has been integrated it in the TFC Local Control Project. Figure 108 is a PVSS panel which displays monitoring and configurable parameters. The FSM state of the devices (for example OdinV2_00) is shown in a button next to the name of the device (RUN_NOT_READY). The initial state of OdinV2_00 is **RUN_READY**. From this state, it can go to RUN_RUNNING. If something goes wrong, it goes automatically to RUN_ERROR. The buttons in grey mean that the device is not part of the data taking (not active). For instance, there is only OdinV2_00 which is active. The other parts ECAL, HCAL, …, VELO are inactive. This kind of run is used to test the readout supervisor OdinV2_00.

Configurable parameters are the ones on the right (which can be ticked, in the L0_trigger frame). They are related to L0 Trigger properties.

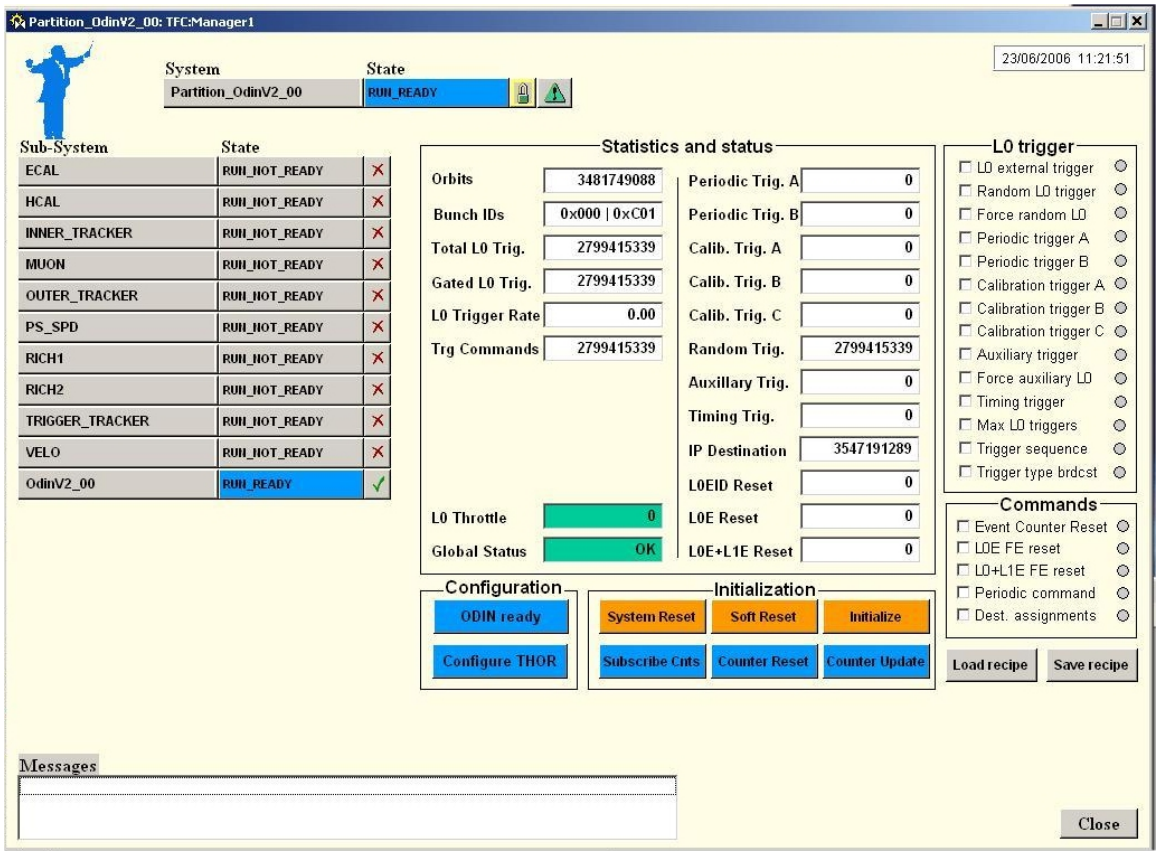


Figure 108. Possible configurable parameters.

Figure 109 shows how a recipe can be saved for OdinV2_00. The current values of the parameters will be saved in the CIC DB.

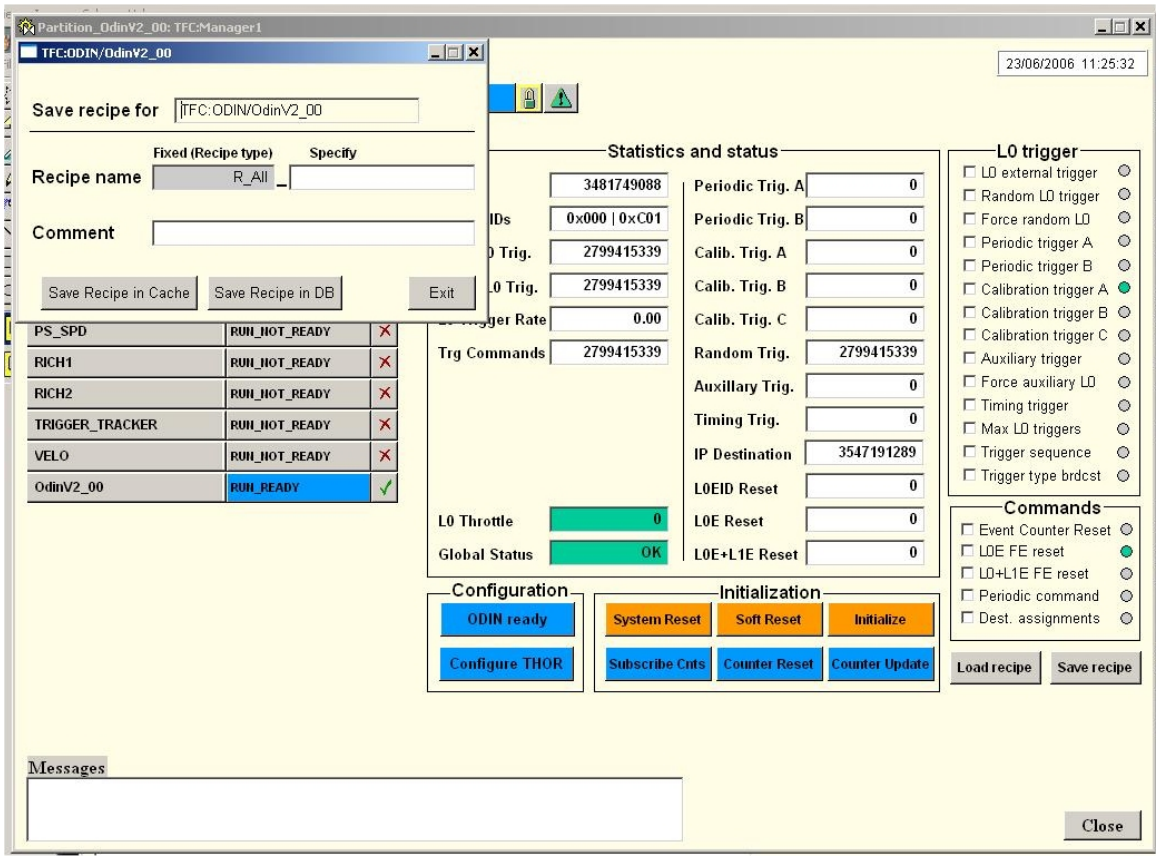


Figure 109. Saving a recipe for OdinV2_00.

Figure 110, Figure 111 and Figure 112 show the different steps to load a recipe into a device. The first thing is to select a recipe and a device name as shown in Figure 110. Then it loads the recipe content into the data point elements. All the parameters have been reset (nothing is ticked) as shown in Figure 111. The state of OdinV2_00 is RUN_CONFIGURING which means the device is being configured. Finally the parameters are applied to the panel (dots in green) as shown in Figure 112. In our case, the recipe modifies the value of the parameter “periodic trigger A”.

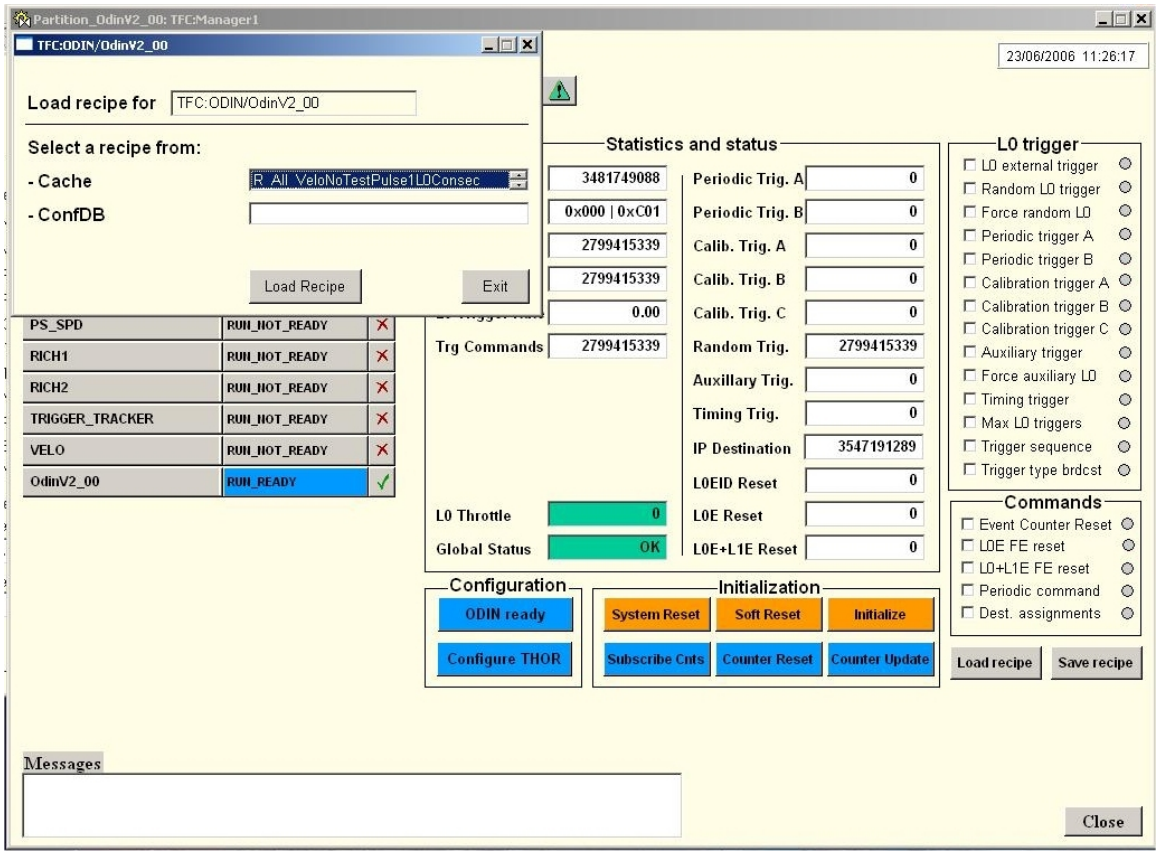


Figure 110. Loading an existing recipe for OdinV2_00.

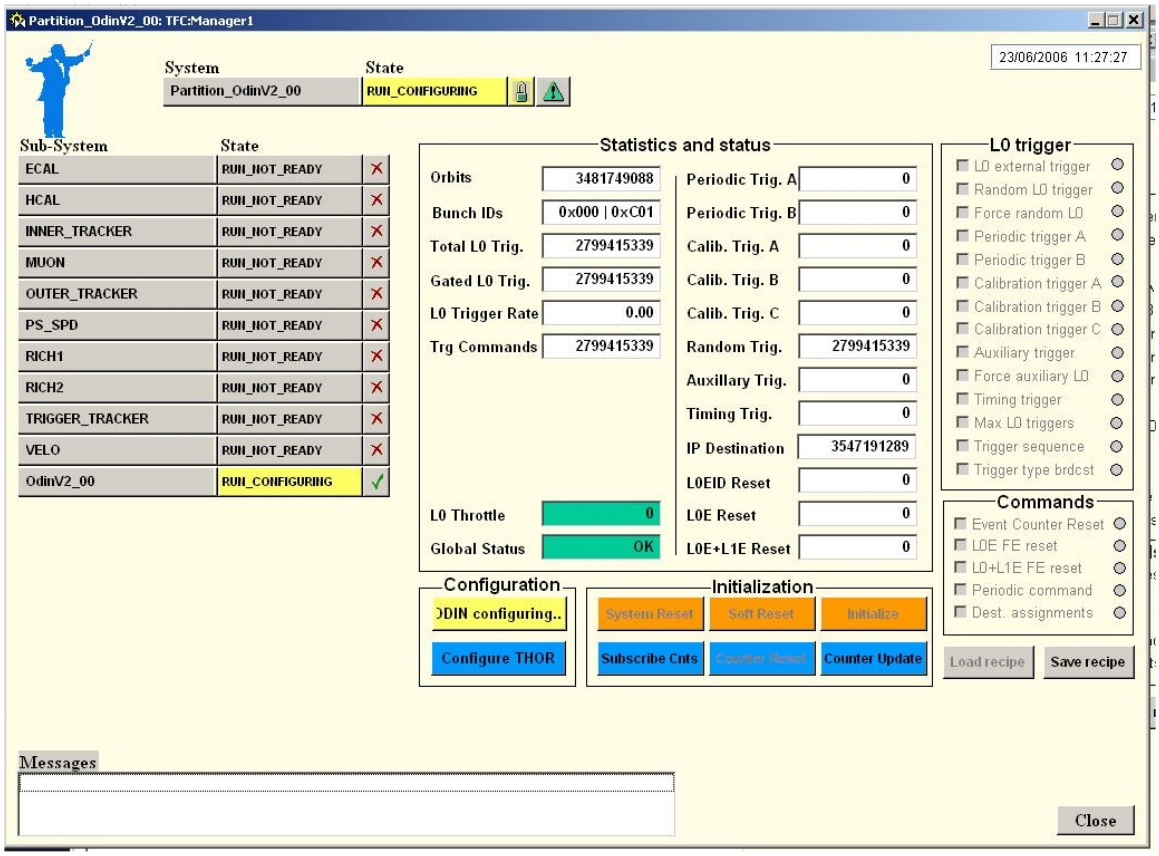


Figure 111. Configuring the hardware.

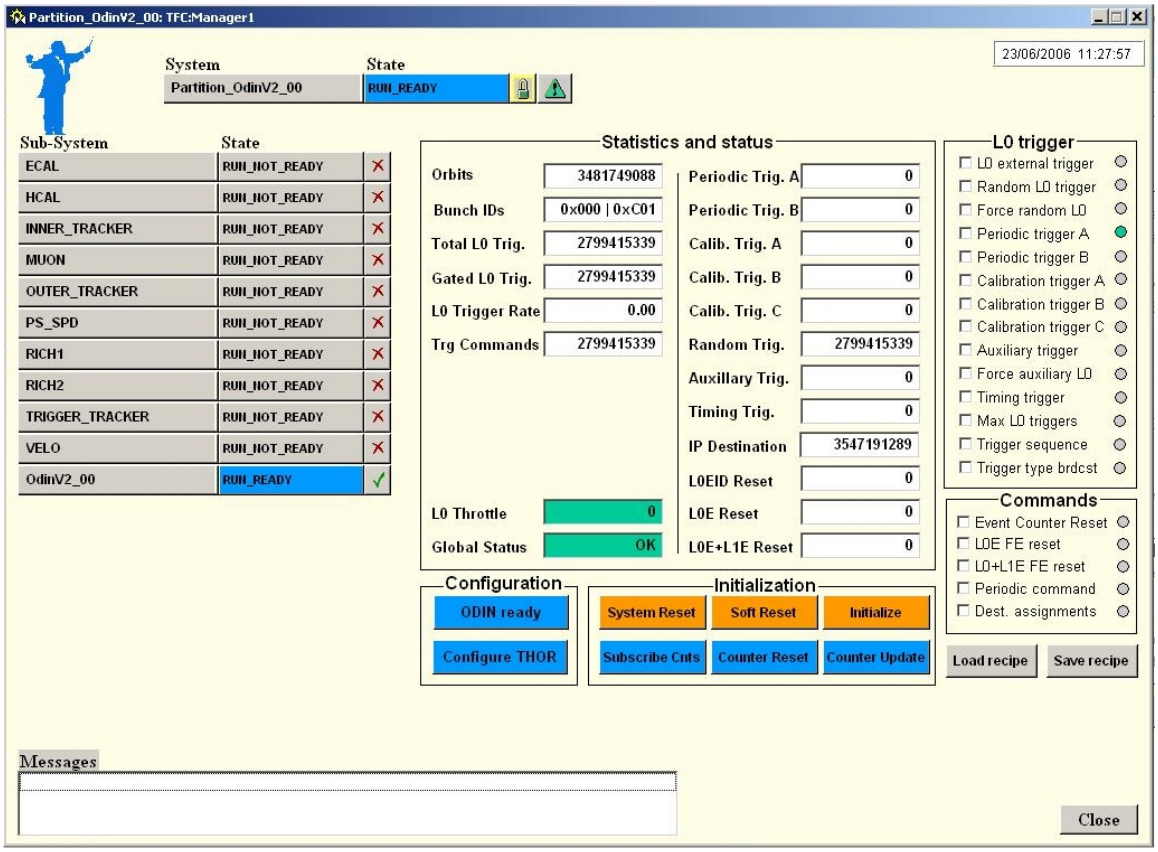


Figure 112. Hardware configured.

So using this panel one can configure the readout supervisor (the real hardware) by loading recipes from the PVSS cache or the CIC DB. If the user is an expert in the readout supervisor, he can also save recipes for it. This panel also allows users to monitor the readout supervisor via different parameters such as Bunch IDs, Trg commands, etc.

Finally the state of the readout supervisor is given by the FSM.

8.3.3 Combining connectivity and configuration parameters

The two previous subsections o shows how connectivity and device parameters for configuration need each other. To define the partition there is a need to get information about the connectivity (referring to Figure 103 and Figure 104), and to set the registers in the partition the information about configuration parameters (referring to Figure 110, Figure 111).

Mixing different types of information is performed at the level of PVSS panels and scripts. It permits the construction of an autonomic control system as PVSS is aware of the changes and can update the different elements as shown in Chapter 7 section 7.1.1.5 and 7.1.2.3. In that case PVSS programs the TFC switch using the connectivity and then configure the readout supervisor according to the running mode.

8.3.4 Displaying the routing table in PVSS

After describing the generation of the routing tables the next section will detail how the generated information is loaded into the switch.

All the hardware must be configured via PVSS. Switches will also be configured using PVSS.

PVSSLoadRoutingTable is a function of CIC_DB_lib to be used to load the routing table of a given switch.

The signature of the function in PVSS is as follows:

```
Int PVSSLoadRoutingTable (string switch_name,
                          dyn_string[14] & destination_name_list,
                          dyn_string & nextport_list,
                          dyn_string & ipnext_list,
                          dyn_string & subnetnext_list,
                          dyn_string & macaddnext_list,
                               string     ErrMess);
```

This function can be called in a PVSS script.

Figure 113 and Figure 114 show PVSS panels to load the routing table of a given switch from the CIC DB to PVSS. Figure 113 displays all the switches. The user selects one switch and pushes the “Load Routing Table” button. Then Figure 114 displays the routing table of the selected switch.

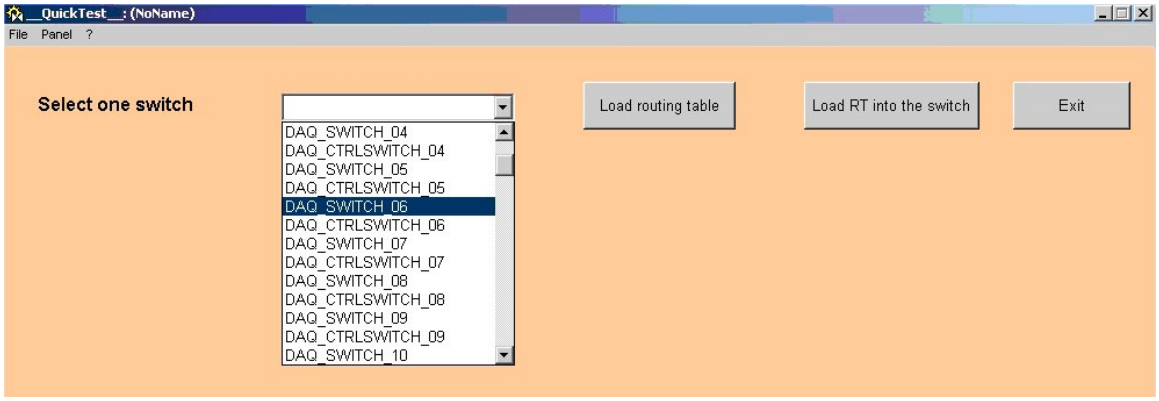
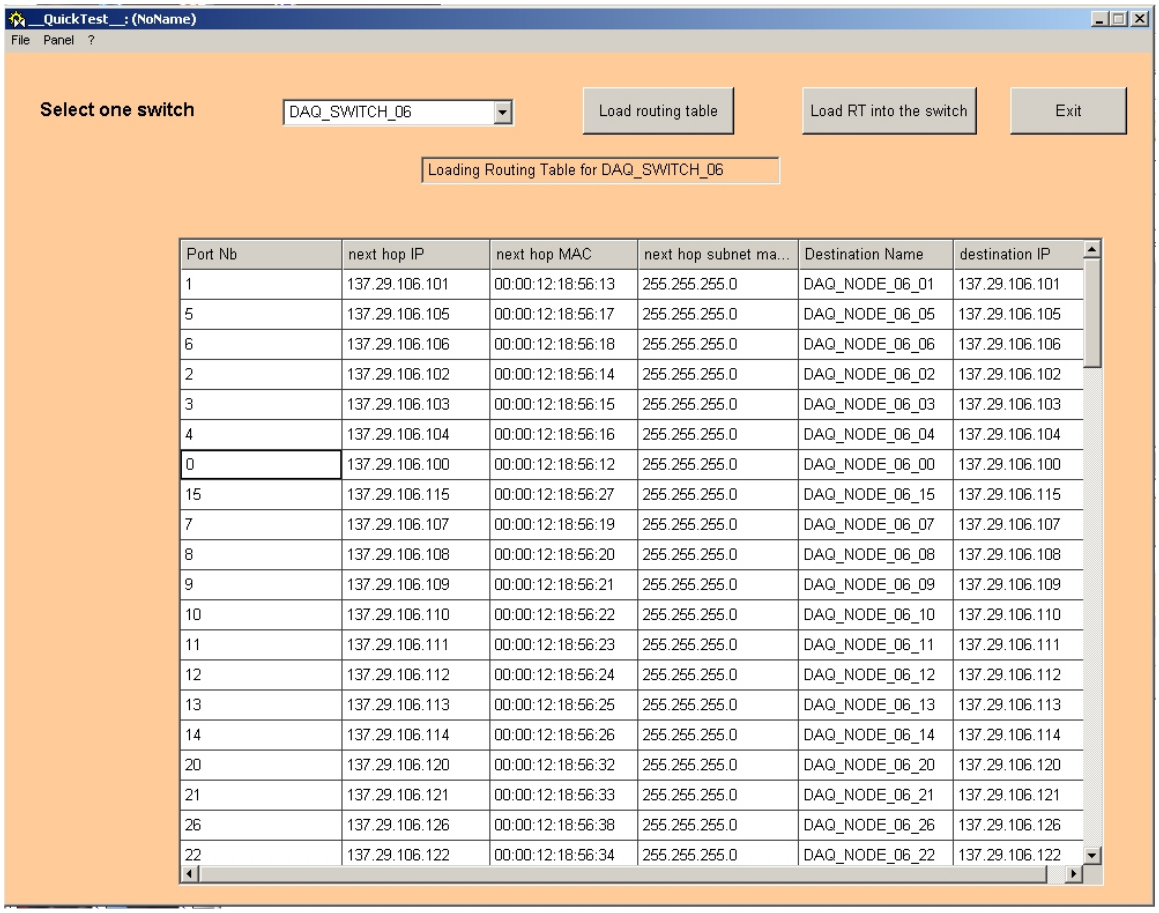


Figure 113. First step: select a switch.

[14] Dyn_string is a type specific to PVSS and it is similar to the vector<string> type, part of STL.



Port Nb	next hop IP	next hop MAC	next hop subnet ma...	Destination Name	destination IP
1	137.29.106.101	00:00:12:18:56:13	255.255.255.0	DAQ_NODE_06_01	137.29.106.101
5	137.29.106.105	00:00:12:18:56:17	255.255.255.0	DAQ_NODE_06_05	137.29.106.105
6	137.29.106.106	00:00:12:18:56:18	255.255.255.0	DAQ_NODE_06_06	137.29.106.106
2	137.29.106.102	00:00:12:18:56:14	255.255.255.0	DAQ_NODE_06_02	137.29.106.102
3	137.29.106.103	00:00:12:18:56:15	255.255.255.0	DAQ_NODE_06_03	137.29.106.103
4	137.29.106.104	00:00:12:18:56:16	255.255.255.0	DAQ_NODE_06_04	137.29.106.104
0	137.29.106.100	00:00:12:18:56:12	255.255.255.0	DAQ_NODE_06_00	137.29.106.100
15	137.29.106.115	00:00:12:18:56:27	255.255.255.0	DAQ_NODE_06_15	137.29.106.115
7	137.29.106.107	00:00:12:18:56:19	255.255.255.0	DAQ_NODE_06_07	137.29.106.107
8	137.29.106.108	00:00:12:18:56:20	255.255.255.0	DAQ_NODE_06_08	137.29.106.108
9	137.29.106.109	00:00:12:18:56:21	255.255.255.0	DAQ_NODE_06_09	137.29.106.109
10	137.29.106.110	00:00:12:18:56:22	255.255.255.0	DAQ_NODE_06_10	137.29.106.110
11	137.29.106.111	00:00:12:18:56:23	255.255.255.0	DAQ_NODE_06_11	137.29.106.111
12	137.29.106.112	00:00:12:18:56:24	255.255.255.0	DAQ_NODE_06_12	137.29.106.112
13	137.29.106.113	00:00:12:18:56:25	255.255.255.0	DAQ_NODE_06_13	137.29.106.113
14	137.29.106.114	00:00:12:18:56:26	255.255.255.0	DAQ_NODE_06_14	137.29.106.114
20	137.29.106.120	00:00:12:18:56:32	255.255.255.0	DAQ_NODE_06_20	137.29.106.120
21	137.29.106.121	00:00:12:18:56:33	255.255.255.0	DAQ_NODE_06_21	137.29.106.121
26	137.29.106.126	00:00:12:18:56:38	255.255.255.0	DAQ_NODE_06_26	137.29.106.126
22	137.29.106.122	00:00:12:18:56:34	255.255.255.0	DAQ_NODE_06_22	137.29.106.122

Figure 114. Loading the routing table.

8.4 Conclusion

In this chapter the GUI layer has been described. Different tools have been implemented. CDBVis is a Python graphical tool which displays the connectivity of devices. It also permits to insert the connectivity.

PVSS panels have been built to configure modules based on the PVSS library for recipes and on the PVSS CIC_DB_lib. It includes displaying the routing tables and handling partitions.

One of the common issues of the GUI layer is to display the information in a user-friendly way. Viewing the connectivity information with all the links was not easy to manage. Improvements can still be done.

With this layer, the database and the object layer could have been tested. Indeed bugs in the PVSS library for recipes and in the CIC_DB_lib could have been discovered and fixed.

Moreover with this type of architecture, one can build a single click control system. The control system based on PVSS can access to any types of information stored in the CIC DB. So PVSS panels can be implemented to modify and load recipes to the devices part of the partition using the connectivity information. The next chapter focuses on the validation of the routing and destination algorithms in PL/SQL.

References

[1] T.Johansen, L.Abadie, E. van Herwijnen, R. Shade, LHCb Configuration Database Visualizer, LHCb Technical Note v1r2, September 2006. LHCB COMP 2006-028.

[2] FwHW tool website, http://lhcb-online.web.cern.ch/lhcb-online/ecs/PVSS_TELL1/default.html.

Chapter 9 Validation of the routing and destination algorithms

The chapter describes the tests which have been carried out to validate the algorithms related to paths (database layer). First we give a brief reminder of the **routingtable_pck** PL/SQL. Secondly we describe the different topologies to which the routing and destination algorithms have been applied and give some measurements, namely the execution of creating a routing table. Thirdly we explain the results of the test and expose the parameters which have an impact on the performance of the algorithm. Then, we show how to load the routing table once it is generated in the CIC DB. Finally we describe the test done to validate the dhcp config file generated by the Perl script “dhcpCfg_generate.pl” by loading it into a DHCP server.

9.1 Outline of the routingtable_pck package (reminder)

As explained in the previous chapter, the routing tables of the DAQ switches, the destination of the DHCP servers and the TFC switch are generated using a PL/SQL package, **routingtable_pck.** This package is only used for the TFC and DAQ connectivity.

Performance for generating these tables is not an issue. Destination and routing tables will be generated and stored in the CIC DB in advance. The information can be loaded whenever it is required. However loading should be very fast.

It is important to note that these predefined tables are stored and maintained by functions in the **routingtable_pck** package. They are executed whenever a change related to the TFC or DAQ systems occur, such as:

- Adding new links between existent or new modules. If the modules are new, they need to be inserted. Rows will be inserted in the **FUNCTIONAL_DEVICES**, **HARDWARE_DEVICES** and **DEVICE_HISTORY** tables. The ports of the new modules need also to be inserted. Rows will be inserted in the PORT_PROPERTIES, **HWPORT_PROPERTIES** and **IPINFO** tables (for DAQ modules only). Then the links between the modules must be inserted. Rows will need to be inserted in the **CONNECTIVITY** table.
- Deleting links. If the link between (device A, port X) and (device B, port Y) does not exist any more, this link should be deleted. So the corresponding will be deleted from the **CONNECTIVITY** table.
- Disabling a functional device. For instance, the farm node FRAM0101 has to be disabled because it does not work properly. Its attribute **FUNCTIONAL_DEVICE .nodeused** will be set to 0. Automatically the links going in and out from the disabled device will be disabled. So the **CONNECTIVITY.lkused** of these links will be set to 0.
- Disabling a link. For instance for debugging reasons some links must be disabled. **CONNECTIVITY.lkused** of these links will be set to 0.
- Updating any of these following attributes for a link: **bidirectional_link_used**, **link_type**, **system_name.**

For the benchmarks, I applied the following method of testing, as recommended by the CERN central DB support group: each query or PL/SQL function was performed 5 times. I took the mean, noted **ā** of the 5 execution times (time interval between queries: 1s -5s – 5mn- 1h) and the standard deviation, noted **σ**. I also displayed the worst and best cases. I used the timing feature of Oracle SQL*plus. It is important to note that this database is accessed by hundreds of users. So some results can change depending on the load on the database when executing the test.

Tests have been performed on my desktop PC (Windows XP Professional version 2002), of almost 3 years old (768MB, Pentium 4 and 2.4 GHz).

9.2 Robustness of the routing and destination algorithms to different topologies

One of the tests was to test the robustness of the routing algorithm to different network topologies.

9.2.1 Destination algorithm applied to the TFC system

A simplified view of the TFC system is shown in Figure 115. It can be considered as a tree with the TFC switch (THOR_00) at the root and the TELL1 boards (represented by SubDet_L1FE) as its leaves. TTCtx, TTCoc and TTCrx are fan-outs, i.e. they split the signal so that it can reach by more devices. SubDet_L0FE contains L0 electronics whose connectivity is represented in Figure 116.

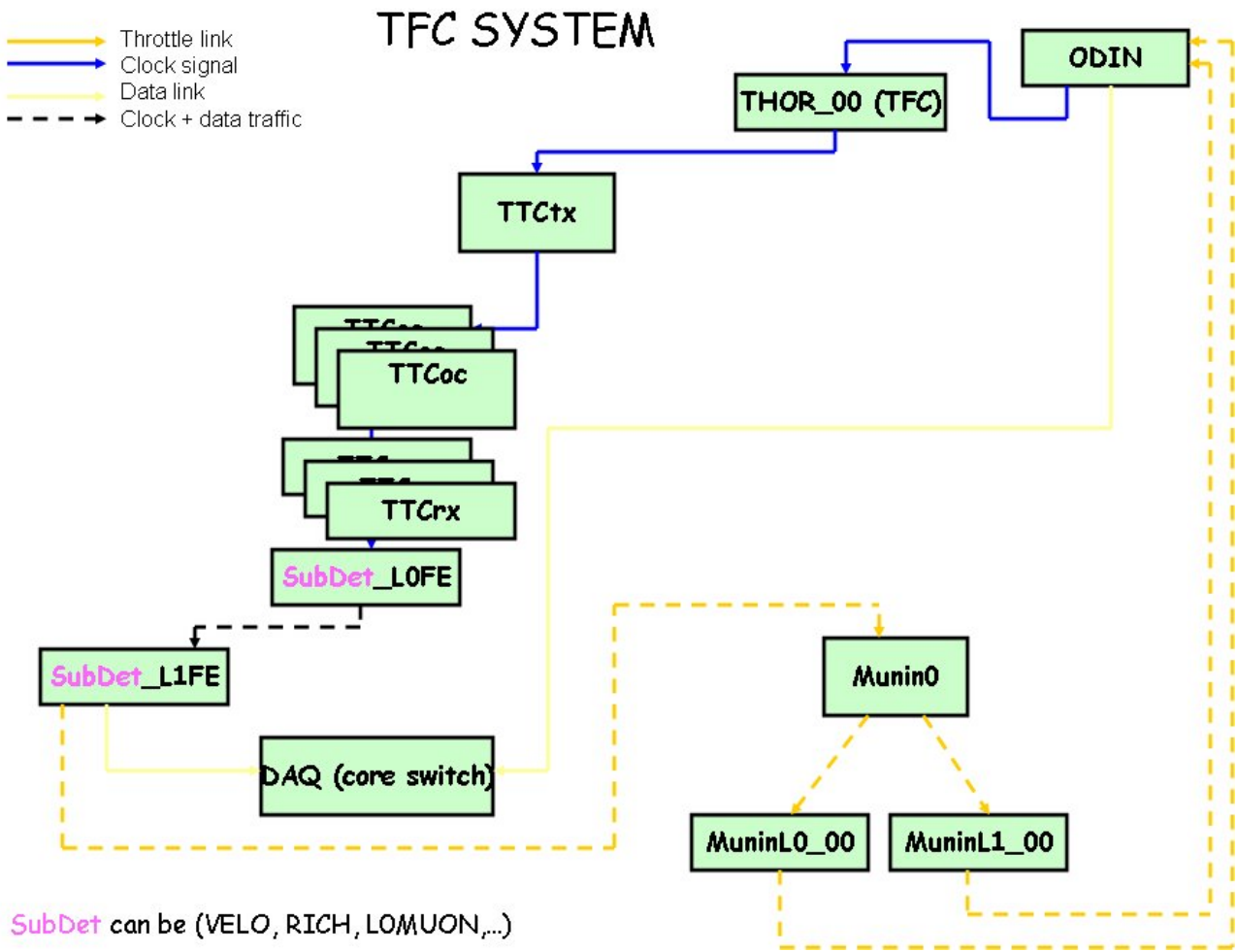


Figure 115. Simplified overview of the TFC system.

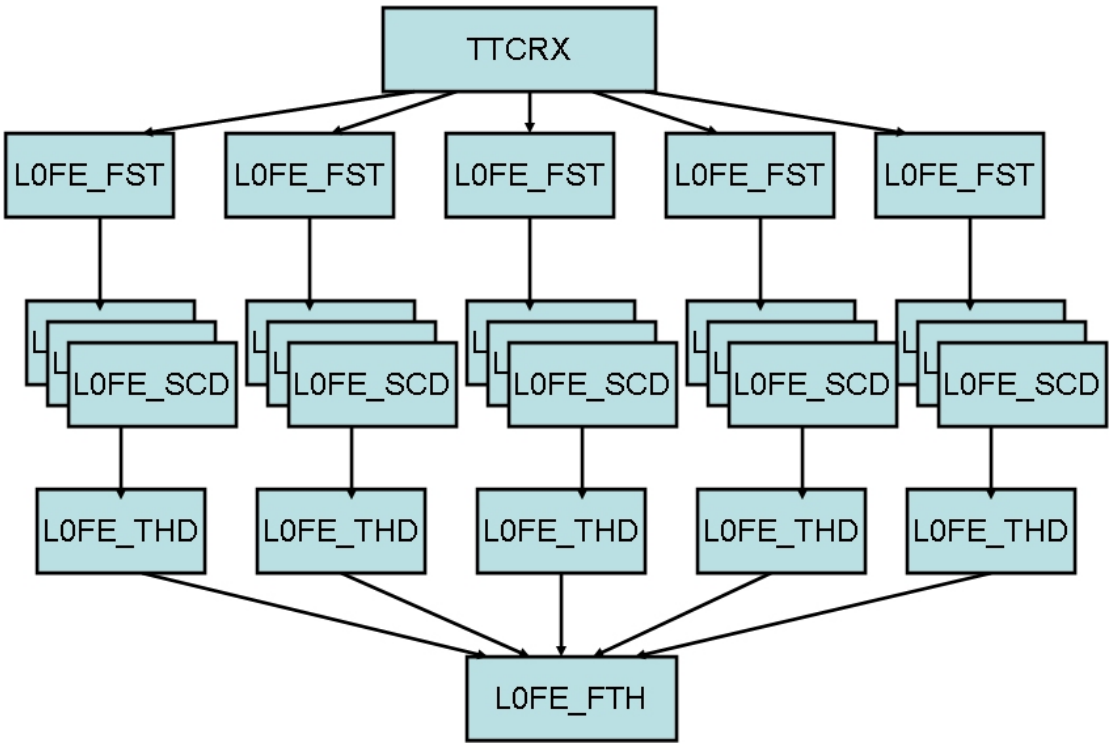


Figure 116. Connectivity for the L0 electronics.

Table 19 shows he number of devices of the given type in the TFC system. The connectivity of the L0 electronics for each subdetector could not be known at the writing time of the PhD. During my PhD studies, I could not get the final connectivity schema of all the subsystems. The only thing I knew that there are around 50,000 modules involved in this connectivity and also 3 levels. So the connectivity for the L0 electronics has been simulated as Figure 116.

Apart from that, the upper part of the connectivity is known.

There is one link between an ODIN and the TFC switch. There is one link between a TTCtx and the TFC switch.

A TTCtx drives 10 TTCocs, and a TTCoc 8 TTCrxs. Each TTCrx is connected to 6 L0FE_FSTs. A L0FE_FST is connected to 5 L0FE_SCDs. A L0FE_THD serves 10 L0FE_SCDs. A L0FE_FTH is connected on its input to 5 L0FE_FTH. The L0FE_FTH are connected to the TELL1 boards. The connectivity between L0FE_FTHs and TELL1 boards depends on the subsystem, as the number of TELL1 boards is different. All the links are unidirectional. The host nodes are the TELL1 boards and the readout supervisors, in italic in Table 19. MuninL1_00 and MuninL0_00 correspond to the Throttle switches which have been explained in Chapter 1. Munin0 is a throttle OR switch (32 inputs *1 output). These three devices are used to alert the readout supervisors in case of buffer overflows. .

Device Type Name	**Number of devices**
ODIN (readout supervisor)	16
THOR_00 (TFC switch)	1
TTCtx	15
TTCoc (Optical coupler)	150
TTCrx	1200
L0 electronics First Layer (L0FE_FST) Second Layer (L0FE_SCD) Third Layer (L0FE_THD) Fourth Layer (L0FE_FTH)	47520 7200 36000 3600 720
TELL 1 boards *VEL0_A* *VELO_C* *PUS* *RICH1* *TT* *IT* *OT_A* *OT_C* *RICH2* *PRS* *ECAL+L0CALO* *HCAL* *MUON_A+L0MUON* *MUON_C+L0MUON* *L0DU*	384 46 45 12 24 48 45 31 38 36 11 13 +1 L0CALO 12 7 + 3 (L0MUON) 7 + 4 (L0MUON) 1
Munin0	15
MuninL0_00	1
MuninL1_00	1
Total Number of devices	**49303**

Table 19. Number of devices per type.

Devicename	**Nb of destinations**	**Nb of paths**	**Execution time (mn)**
TFC switch	384 (= nb of TELL1)	36000 (equal to the number of L0FE_SCD)	Try 1 : 1mn31s07 Try 2 : 1mn07s04 Try 3 : 1mn03s06 Try 4 : 1mn05s06 Try 5 : 1mn31s03 **ā**: 1mn15s06 **σ**: 0mn15s

Table 20. Result of the execution time of the destination table of the TFC switch.

The generation of the destination table takes more than one minute.

9.2.2 Routing algorithm applied to the Flower topology

Figure 117 represents the old design of the DAQ network (until summer 2005). The shortest path should always be selected if several routing paths to a same destination are found. This topology was very useful to check that the algorithm was robust against cycles (i.e. the first node is equal to the last node in the path) [1].

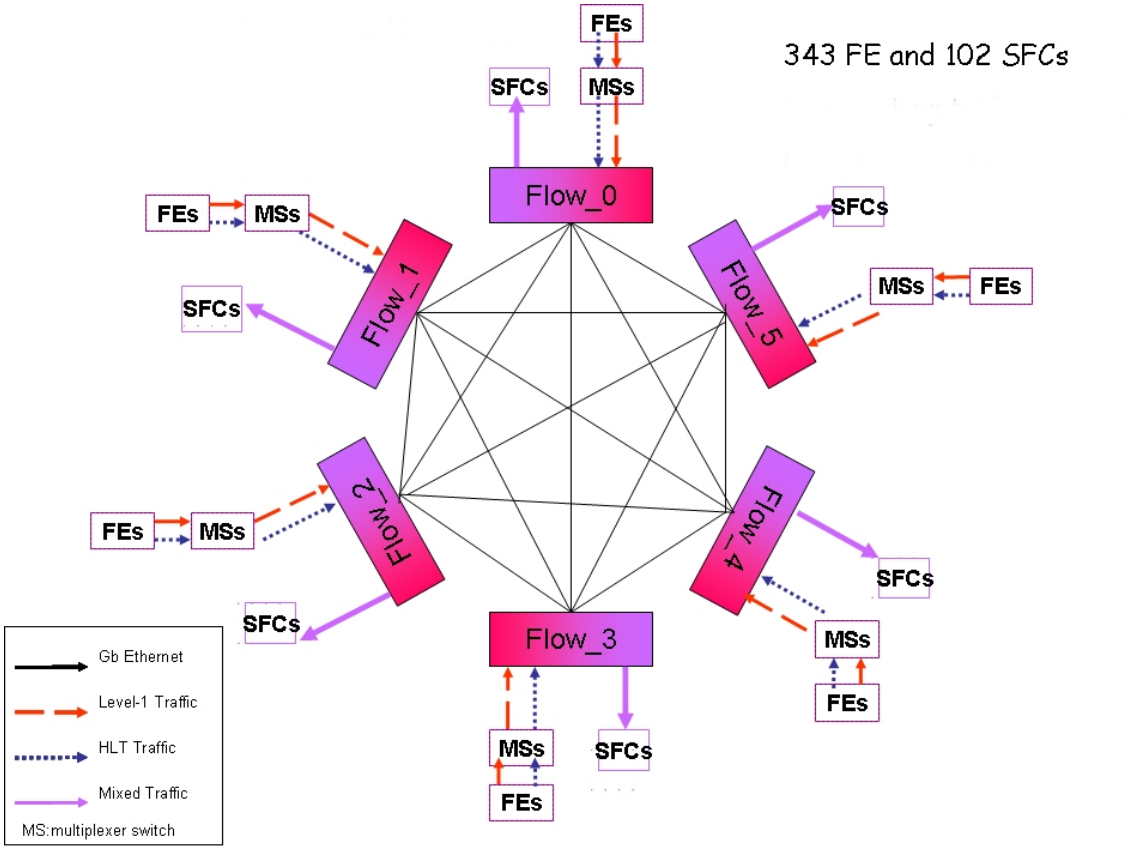


Figure 117. Flower topology.

Referring to Figure 117, data is coming out from the FEs (there are TELL1 boards from the different subdetectors) to the MSs (there are switches). Each FE (340 sources) is connected to a port of one MS (12 Multiplexer switches).

Then the data is multiplexed and go to the Flow devices. There are 6 Flow devices which are fully connected. Each MS is connected to a port of a Flow switch. The links between them are bidirectional. In the DAQ system, these links are the only ones which are bidirectional. Then the data goes from the Flow devices to the SFCs (SubFarm Controllers, PCs which rebuild the event). Then each SFC (102 sub-farm controllers) is connected to a port of a Flow switch. In other words, each Flow has 17 output ports connected to 17 SFCs (17*6=102). All the links carry data traffic (this is the link type).

The host nodes are FEs and SFCs in Figure 117.

Each SFC is connected to a sub-farm switch which is connected to 20 sub-farm nodes (not shown in the picture).

The routing tables of Flow_1 … Flow_5, MSs have been generated with success.

Table 21 gives the results of the tests. We have verified mathematically that the number of paths found is correct. In our case, we could exploit some properties of the graph. For instance, for the Flower topology, to count the number of paths between a

Flow device and a device of type SFC, permutations are used (see Appendix J for the proof). Figures in red represent the min and max values of the execution time.

Devicename	**Nb of destinations**	**Nb of possible paths found**	**Execution time (in sec.)**	**Nb of tries (out of 5)**
Flow_0	102 (only the SFCs)	5542	19.02 5.04 6.05 6.09 22.02 **ā**: 11.64 **σ**: 7.32	1 2 3 4 5
Flow_3	102 (only the SFCs)	5542	13.06 7.00 5.05 5.09 18.02 **ā** : 9.64 **σ** : 5.12	1 2 3 4 5
Flow_5	102 (only the SFCs)	5542	9.05 8.06 9.03 10.08 21.00 **ā :**11.44 **σ** : 4.83	1 2 3 4 5
MS_03	102 (only the SFCs)	5542	5.02 6.02 6.05 6.01 13.09 **ā :** 7.24 **σ** : 4	1 2 3 4 5
MS_10	102 (only the SFCs)	5542	3.09 4.09 4.09 4.09 8.07 **ā :** 4.68 **σ** :1.75	1 2 3 4 5

Table 21. Summary of the execution time of routing tables.

What is essential is to ensure that the routing tables generated are correct and consistent. When the run starts, the ECS will load the routing tables from the CIC DB to PVSS and from PVSS to the switches. As the routing tables have been already generated, the loading is much faster see section 4.

9.2.3 Routing algorithm applied to the DAQ topology

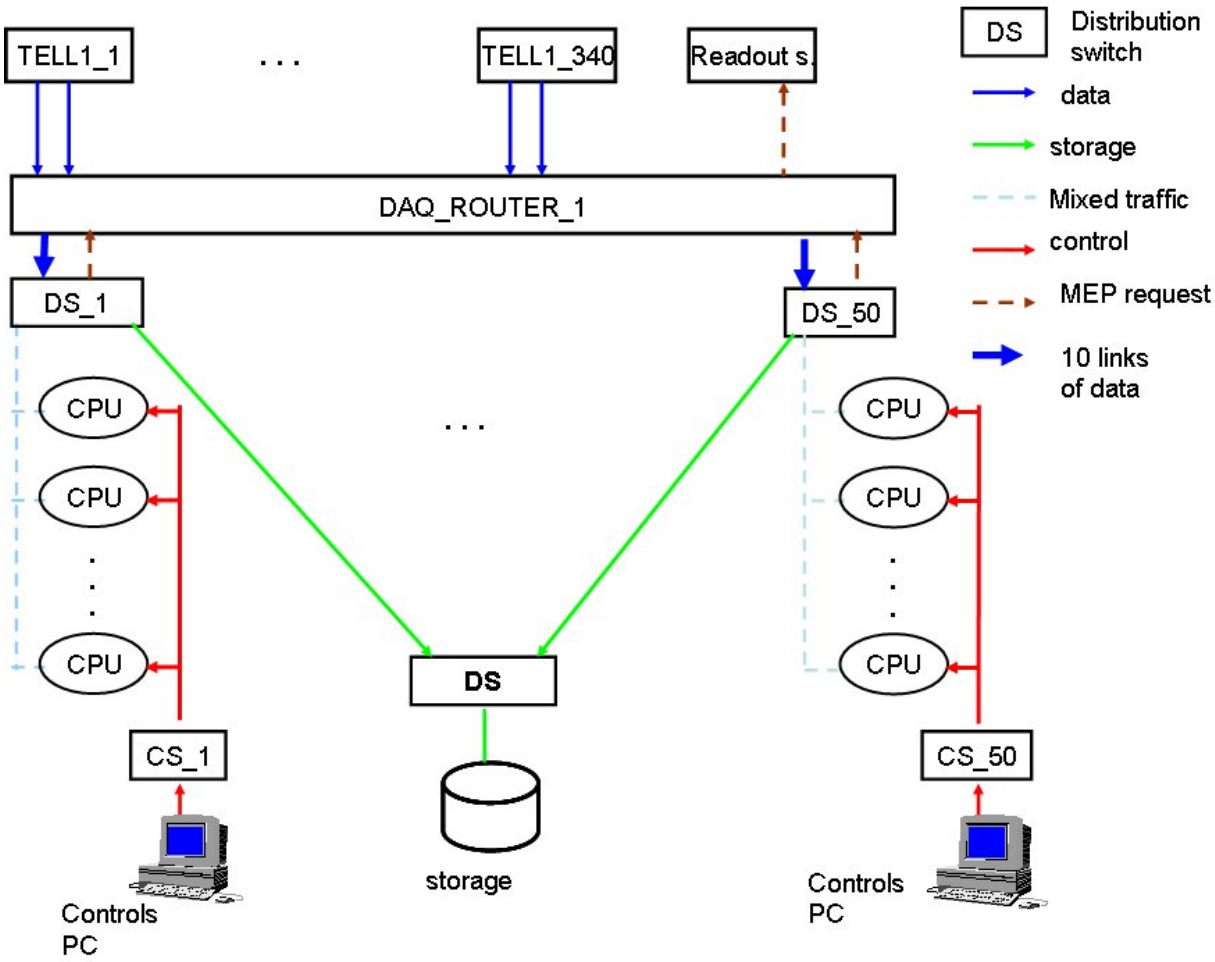


Figure 118. The foreseen topology of the DAQ. Links which are not arrow are bidirectional links.

Figure 118 describes the foreseen connectivity of the DAQ system. There are different types of links (data, storage, control, mep_request traffic and mixed_traffic which combines both mep_request and data traffic). The routing tables of the Force Ten - also called DAQ_ROUTER_1- ,DS_1 to DS_50 and DS and CS_1 to CS_50 (Control Switch) should be generated.

The host nodes are the farm nodes (Trigger Farm + Local Storage), the TELL1 boards, the readout supervisors and the controls PCs.

In the tests, there were 329 TELL1 boards connected twice to DAQ_ROUTER_1. Then there were 10 links between DAQ_ROUTER_1 and each of the 50 DS (Distribution Switch). Each DS is connected to 40 farm nodes. There is one link in brown (referring to Figure 118) between DS and DAQ_ROUTER for MEP request, for the readout supervisor. There are 16 readout supervisors which are connected to the DAQ_ROUTER_1. The storage has been represented as sub-farm with 50 PCs (represented with a cylinder). The TELL1 boards are controlled by controls PC via Control Switches, but it is not shown in Figure 118.

Only the links between DSs and farm nodes are bidirectional, the other ones are unidirectional. There are four types of links, data, control, storage and mep_request. Only the links between DSs and farm nodes carry a mixture of traffic, storage, mep_request and data.

With this topology, I verified that my algorithm produced consistent routing tables. Consider the routing table of the DAQ_ROUTER_1. The possible destinations are the

2000 farm nodes and the 16 readout supervisors. To send a packet to a given farm node there are 10 possible paths as there are 10 links from DAQ_ROUTER_1 to a given DS. However, only one out of the 10 links should be selected, otherwise the routing table is inconsistent as the router will not know to which output port it should forward the packet. The link type compatibilities are also been checked. For instance, none of the farm nodes of the local storage are in the routing table of DAQ_ROUTER_1 as data traffic is incompatible with storage traffic.

Devicename	**Nb of destinations**	**Nb of possible paths found**	**Execution time (in sec.)**	**Nb of tries (out of 5)**
DAQ_ROUTER_1	2000 + 16 (the farm nodes of the EFF + 16 readout sup.)	2016 (without taking into account port) 20016, with ports	15.06 12.06 12.02 13.02 15.09 **ā**: 13.45 **σ** : 1.37	1 2 3 4 5
DS_1	40 +50 + 16 (the farm nodes + PCs part of the storage + readout supervisors)	106 in both cases	3.02 2.02 3.01 3.02 4.04 **ā** : 3.02 **σ** : 0.65	1 2 3 4 5
DS_40	40 +50 + 16 (the farm nodes + PCs part of the storage + readout supervisors)	106 in both cases	3.08 3.04 3.01 3.02 4.00 **ā** : 3.23 **σ** : 0.38	1 2 3 4 5
DS	50 (only the storage PCs)	50 in both cases	3.00 2.06 2.08 5.02 8.01 **ā**: 4.03 **σ** : 2.27	1 2 3 4 5
CS_20	40 (only farm nodes)	40 in both cases	3.05 4.03 3.02 3.00 8.07 **ā**: 4.23 **σ** : 1.96	1 2 3 4 5

Table 22. Result of the execution of routing tables for switches of the DAQ foreseen topology.

9.2.4 Generating all the routing tables for the DAQ system

Another test was to generate all the routing tables of the DAQ system in one go, sequentially. So I wanted to measure the execution time of 50+50+1+1=102 routing tables, resp. DS, CS, DAQ_ROUTER_1 and DS_STORAGE.

Here again, I have repeated 5 times the test using a PL/SQL script. It has been executed from SQL*plus. The results are presented in Table 5.

Try	Execution time (sec)
1	63
2	79
3	96
4	84
5	86
	ā: 81.6
	σ: 10.2

Table 23. Execution time for generating routing tables.

N.B: in the DAQ system, all the intermediate nodes correspond to devices of type switches or routers. **FUNCTIONAL_DEVICES.node** is the attribute which permits to distinguish a host node from an intermediate node. It is set to 1 if it is a host node and 0 if it is an intermediate node. So to select all the DAQ switches we just select devices which belong to the DAQ system and **FUNCTIONAL_DEVICES.node**=0.

The advantage of generating the routing tables sequentially is that the first part of the algorithm is performed only once. Indeed as all the switches are in the same system, i.e. DAQ, so the simplified view of the connectivity in which the ports are removed (see Chapter 6 section 6.2.2, is the same. This view does not depend on the switch. So the **AGGREGATED_LINKS** table is the same if we generate the routing table for DAQ_ROUTER_1 or DS_1, for instance.

The same remark is done for grouping links per pair. This step is independent from the switch. So the **LINK_PAIRS** table has exactly the same content regardless of the switch given in input. So there is no need to repeat this operation 102 times (there are 102 switches so 102 routing tables to be created), once is sufficient. That is why the execution time of generating all the routing tables is not very high. The first part takes most of the execution time as it is explained in section 3.3.

Using this remark, these two tables can be created once, when after a change in the DAQ network, all the routing tables need to be updated. It spares time.

9.3 Comments on the test results

9.3.1 Robustness

Using the DAQ flower topology, I could check that the routing algorithm was robust against cycles and produced consistent routing tables.

Also I verified that the destinations and the number of possible paths found were correct.

Even if it is very unlikely that creating a routing table is performed concurrently, I have tested this situation. It worked in the sense that the routing tables were created

properly and no SQL statements failed. For instance in one of the versions, I used sequences for the **pathid**. A sequence was created whenever a new switch as input parameter was given. If the sequence already exists, I set it back to 1. This statement fails if executed for the same switch at the same time.

9.3.2 Performance optimization

The performance of generating routing tables was optimized together with the central database support on the basis of detailed execution trace files.

The generation of 6 routing tables took 3mn34 sec. We achieved an improvement of a factor of 4, after optimization. In **routingtable_pck** PL/SQL package, there is an extensive use of:

- Bind variables (it allows producing generic SQL statements and it reduces the parse execution time [2]);
- Temporary tables as temporary storage of the results (the main advantage is the content of these table is private to the session);
- Native Dynamic SQL instead of the DBMS_SQL package;
- Avoid functions in the WHERE clause if possible to allow index usage;
- Join instead of IN : for instance use of

```
"select t.devicename from FUNCTIONAL_DEVICES t,
FUNCTIONAL_DEVICE_TYPES e where t.devicetypeid=e.devicetypeid and
e.devicetype like 'DAQ_SWITCH_4_' ;"
Instead of:
"select devicename from FUNCTIONAL_DEVICES where devicetypeid in
(select devicetypeid from FUNCTIONAL_DEVICE_TYPES where e.devicetype
like 'DAQ_SWITCH_4_') ;"
```

These tuning issues enable to improve the performance when generating routing tables sequentially. The SQL queries are cached in memory and data blocks already used are searched in memory instead of physical memory.

9.3.3 Analysis of the routing algorithm by parts

The algorithm has four parts (see Chapter 6 section 6.2.7) which have different contributions to the total execution time. In my previous tests, the average execution time taken by each of these four parts is shown in Table 24**.** I used the **dbms_utility,** an Oracle package to measure the time taken by each of the four blocks.

An analysis of the results shows that:

- For switches in {Flow_0, Flow_3, Flow_5, MS_03, MS_10}, the last part, i.e. inserting everything in **PATH_LINES** and **ROUTING_TABLE,** takes most of the execution time (around 45.7 %). The second part, i.e. finding all the paths, is the fastest. It represents around 6% of the execution time. The first (filling the **AGGREGATED_LINKS** and **LINK_PAIRS**) and the third (mapping with **portid** and selecting one routing path per pair of [destination, network interface]) parts represent 19.5% and 28.8%, respectively.
- For switches in {DS_1, DS_40, DS, CS_20}, the first part takes most of the execution time with 83.1%. The second part is still the fastest with 2.2%. The third and fourth parts represent 11.1% and 3.6% respectively.

- DAQ_ROUTER_1 is special. The second part is still the fastest (0.7%). The third part takes the most of the execution time with 44.6%. Then it is the fourth part with 34.8%.

The second part is always the fastest to be executed. It can be explained by the fact that the SQL statements in this function are against **LINK_PAIRS** and **PATH_LINES_TEMP** tables which have some thousands rows (never more than ten thousands rows), so the table size is small. Then the fact that the second part takes more time for {Flow_0, Flow_3, Flow_5, MS_03, MS_10} is due to the maximum path length. The maximum path length for these devices is respectively {6, 6, 6, 7, 7} whereas for {DAQ_ROUTER_1, DS_1, DS_40, DS, CS_20}, the maximum path length is {2, 2, 2, 1, 1}.

Devicename	**Block**	**Distribution time in %**
Flow_0	1	22.8
	2	5
	3	25.7
	4	46.5
Flow_3	1	13.9
	2	5.9
	3	26.4
	4	53.8
Flow_5	1	18.7
	2	5.5
	3	28.3
	4	47.5
MS_03	1	21.8
	2	7.2
	3	32.8
	4	38.2
MS_10	1	20.3
	2	6.5
	3	30.8
	4	42.4
DAQ_ROUTER_1	1	19.9
	2	0.7
	3	44.6
	4	34.8
DS_1	1	82.7
	2	1.5
	3	12.1
	4	3.7
DS_40	1	82.5
	2	1.6
	3	13.1
	4	2.8
DS	1	82.8
	2	3.3
	3	10.2
	4	3.7
CS_20	1	84.4
	2	2.6
	3	9.1
	4	3.9

Table 24. Distribution of the average per part.

The third part has the highest execution time for DAQ_ROUTER_1. This is because of the ten links between DAQ_ROUTER_1 and DAQ_DS_XX. So the number of paths found with the second function is multiplied by ten. In the other cases, there is only one link between devices. So there are fewer rows inserted in **ROUTING_TABLE_TEMP** than for DAQ_ROUTER_1 (less than 150 against more than 20000). Also, in this step one routing valid path is selected by distinct pairs of [destination, network interface]. The SQL statement is faster when there is only one choice and when the number of destinations is smaller.

The fourth part takes most of the execution time for {Flow_0, Flow_3, Flow_5, MS_03, MS_10}. This function inserts and deletes the biggest number of rows with CHECK options (primary key, foreign key, etc.), as temporary tables have no check options. Even for DAQ_ROUTER_1 this part is quite time consuming but less than the third because of the routing path selection. There are a lot of rows which are inserted during these two parts.

The first part takes most of the execution time for {DS_1, DS_40, DS, CS_20} because compared to the other functions; the number of rows processed is smaller. Indeed the first part inserts around 4550 links in AGGREGATED_LINKS and around 50 in **LINK_PAIRS** whereas the other functions inserts less than 150 rows.

To sum up, the execution time of creating a routing table depends on:

- The number of aggregated links (or logical links) in the system connectivity (the first part is influenced);
- The maximum path length (second part);
- The number of redundant links for the first and last links in the routing path found (third part);
- The number of distinct pairs [destination, network interface] (third part).

The higher these parameters, the slower the execution time is.

9.3.4 Analysis of the destination algorithm by parts

The second part is not the fastest because of the maximum path length, equal to 8.

The first part is faster as there is no bidirectional link, so the number of insert statement is reduced.

Devicename	Part	Distribution time in %
THOR_00	1	12.36
	2	19.39
	3	39.09
	4	29.16

Table 25. Distribution of the average time per part for the TFC switch.

The third part is the highest because of the complex insert statement, similar to the routing tables. The fourth part is still higher because of the total number of paths (36000).

The number of nodes and links for the TFC system is higher than in the DAQ system, hence a higher execution time. The maximum path length is also higher.

9.3.5 Parameters which have an impact on the different steps of the algorithm

The execution time of a routing or of a destination table is not fixed. It depends mainly on the following parameters:

- The number of links stored in the connectivity table. A link should be counted twice if the link is bidirectional. These two numbers have an impact on the first part of the algorithm which fills the AGGREGATED_LINKS and LINK_PAIRS tables.

  In the TFC system there are a lot of links roughly 85309 links (no bidirectional link) stored in the connectivity table. In the DAQ system, there are roughly 5384 links + 2000 bidirectional links for the current version and around 2611 links + 15 bidirectional links for the Flower topology. It takes 3.48 sec in average to perform it for the DAQ flower topology, 3.54 sec for the DAQ current topology and 9.88 sec for the TFC system. For the HCAL connectivity composed of 13952 unidirectional links, it takes 4.3 sec. Figure 119 shows that the execution time is a linear function of the number of rows (equation $y=7.82*10^{-5}x+3.2$). It takes roughly 3 sec to execute the SQL statements which return no rows. In that case both tables AGGREGATED_LINKS and LINK_PAIRS are empty. We could check the result by inserting a functional device which is not connected and part of a fake subsystem.

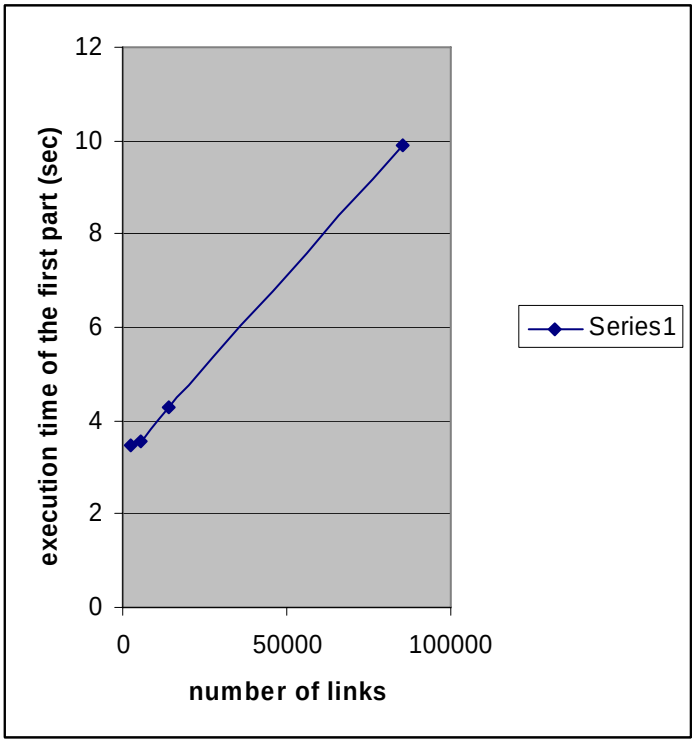


Figure 119. Impact of the number of links on the execution time.

  The number of links has also an influence on the second part of the algorithm as the LINK_PAIRS table will be contain more rows.

- The maximum path length is also important, especially for the second part of the algorithm which finds the valid paths. This parameter is not obvious to quantify as it depends on the number of links and also on the number of paths grouped by

path length. Table 26 shows the impact of the maximum path length. The longest path found is in the TFC system. Also this system contains the most number of paths.

	Length<=3 sec/nb of paths	Length=4 sec/nb of paths	Length=5 sec/nb of paths	Length=6 sec/nb of paths	Length>=7 sec/nb of paths
DAQ_FLOW	0.05 sec/ 123+340	0.10sec/1020	0.13sec/2040	0.14sec/2040	
DAQ_ROUTER	0.07 sec/2016				
TFC_SWITCH	0.26 sec/0	0.58sec/0	3.18sec/0	9.36sec/0	21.58/36000
HCAL_PMT	0.01sec/3				

Table 26. Influence of the maximum path length.

9.4 Loading the routing table into a switch

After describing the generation of the routing tables the next section will detail how the generated information is loaded into the switch.

In chapter 8, displaying the routing tables in PVSS has been explained. Some figures about the execution time and how to load the routing table into a switch are exposed.

9.4.1 Loading the routing tables from the CIC DB to PVSS

It takes roughly 0.125s to load a routing table of a distributed switch from the CIC DB to PVSS, on Windows and on Linux, it takes roughly 0.075s (see Figure 120). Loading the routing table of DAQ_ROUTER_1 requires more time, 0.875 s on Windows and 0.458 s on Linux. One reason may be that PVSS runs faster on Linux than on Windows as it uses a lot of sockets to communicate between the different managers.

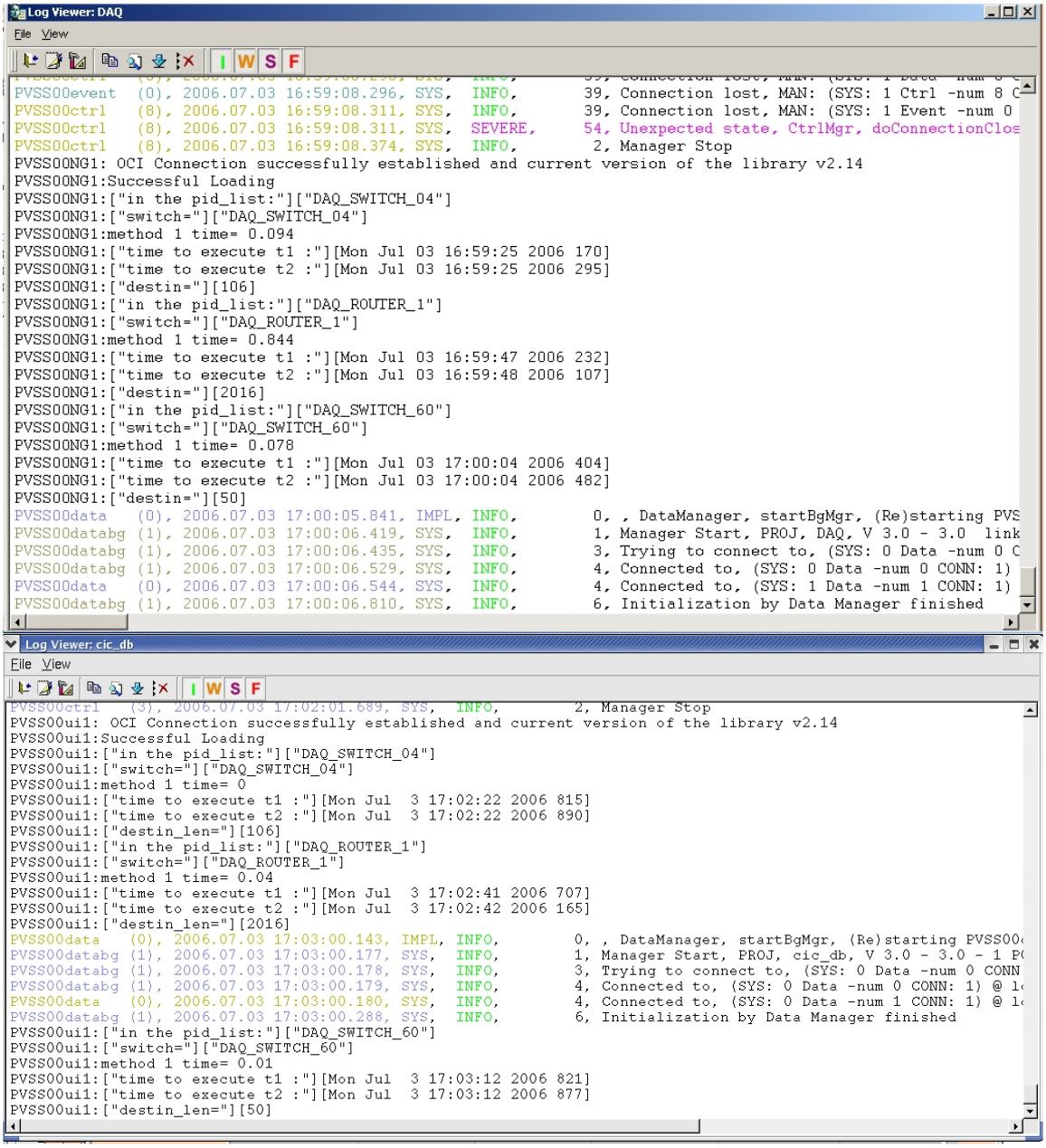


Figure 120. Execution time to load a routing table from the CIC DB to PVSS. The output at the top is from Windows, the one at the bottom, from Linux.

9.4.2 Loading the routing tables from PVSS to a physical switch

There are several ways to load a routing table into a switch/router. It depends a lot on the type of switches. However the principles are the same:

- Open a telnet communication via the port 23 of the switch, using a socket;
- answer the questions asked by the switch to program the routing table ;
- write a program to insert the routing table into the switch, knowing the order of the questions.

As a proof of the concept, I wrote a server to load and check the loading of a routing table from PVSS to a switch.

Figure 121 illustrates the principles of the implementation. Using the CIC DB PVSS library, the routing table of a selected switch can be loaded into PVSS.

Then the routing table is loaded into the selected switch (DAQ_SWITCH_06, in the example) via the RT_server which is a DIM server. The RT_server receives the routing table from PVSS via DIM, a protocol [3] which is lighter than CORBA [4]. Then the RT_server opens a socket on the telnet port of the switch and inserts the routing table as if it was in front of a command line.

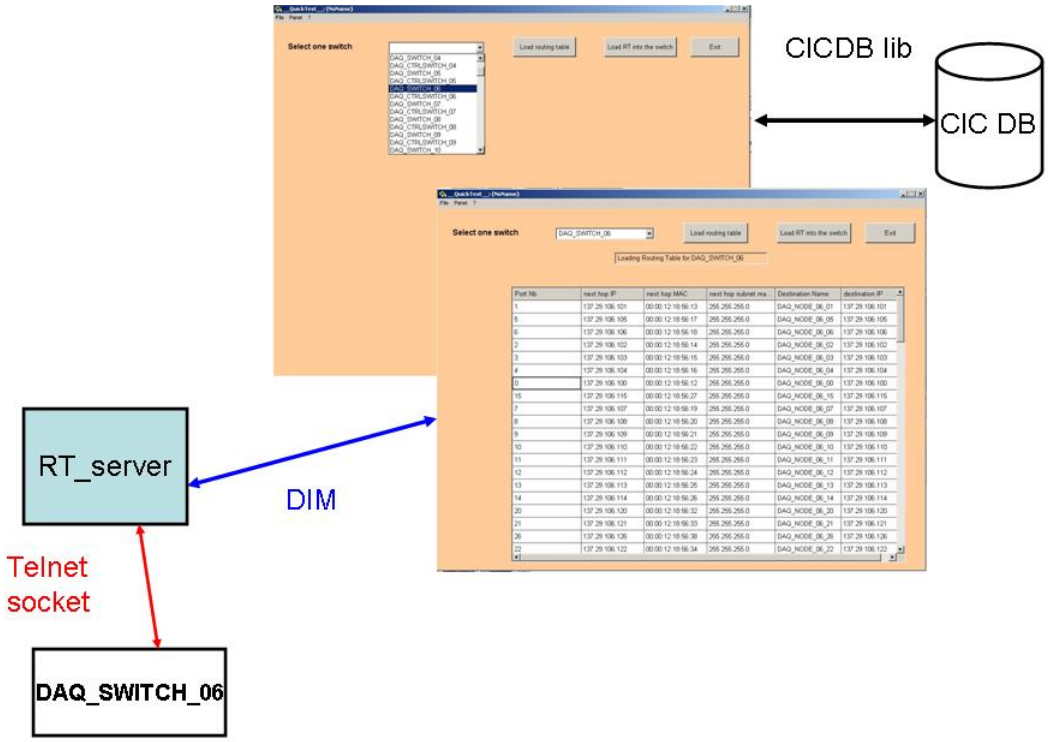


Figure 121. Principles of loading a routing table into a switch.

This part has to be adapted by the network team as the implementation depends on the type of switch. This setup can be extended to be autonomic. The user changes the network from some PVSS panels. Then the routing tables are updated automatically using the **routingtable_pck** PL/SQL package. Then using DIM and the RT_server the routing tables are programmed in the switches of the DAQ.

9.5 Generation and loading the dhcp config file into a DHCP server

Here also performance is not really an issue as the dhcp config files will also be generated “offline”. The essential point was to test that the IP address assignment to the DAQ hosts obtained by creating the dhcp config file was correct and is accepted by the DHCP server.

9.5.1 Prerequisites

To allow automatic creation of a DHCP config file, the following components are required:

- Linux operating system;

- Perl;
- Perl DBI module to query the configuration database;
- Perl XSLT module to allow the use of XSLT functionalities;
- Connectivity with the IP information and boot image information tables filled accordingly in the CIC DB;
- Tns_names.ora to locate the CIC DB.

9.5.2 Usage

First the user inserts the generic options in “dhcp_options.xml”.

Then, in a command line, the user types “perl dhcpCfg_generate.pl <dhcp_server_name>” as shown in Figure 122. In the example, we generate the dhcp_conf for DAQ_CTRLPC_08.

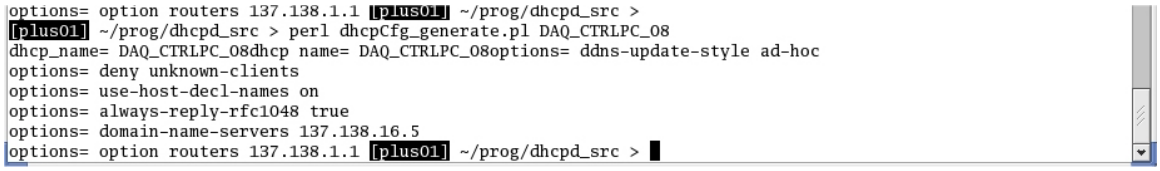


Figure 122. Generating the dhcp config file from a cmd line.

The dhcp_file.xml and an extract of the dhcpd.conf are shown in Appendix C.

It takes in average 40 sec to generate the “dhcpd.conf” file. There are 40 host nodes which get their IP addresses from DAQ_CTRLPC_08 as one can expect it.

Then with the help of the network team, the “dhcpd.conf” produced by this application has been copied to etc/dhcpd.conf of the dhcp_server. The file has been accepted and the test was successful.

The other test was to exclude 10 farm nodes and check that they have disappeared from the “dhcpd.conf”. Farm nodes from ‘DAQ_NODE_08_00’ to ‘DAQ_NODE_08_09’ have been excluded, i.e. **FUNCTIONAL_DEVICES.nodeused**=0. The farm nodes excluded are not in “dhcpd.conf” any longer.

9.6 Conclusion

This chapter has presented the tests carried out to verify the robustness of the routing and destination algorithms. Performance on the execution to create these tables was not an issue. It takes between 5 sec and 1mn15 sec to generate a routing table or a destination table depending on the router or server and the topology of the system. The algorithm depends on two parameters, the maximum path length (less than 10) and the number of links of the connectivity. Verifying the correctness of the routing and destination tables was essential to ensure events data routing from the TELL1 to the farm nodes and to configure the network equipment with IP addresses.

The routing and destination tables are automatically maintained by the **routingtable_pck** PL/SQL package. No human intervention is required to update the content of the tables as the package has been implemented following the autonomics guidelines.

Loading these tables from the CIC DB to PVSS should be fast, less than a few seconds. It takes between 0.075s and 0.875 s to load the routing tables, depending on

the size of the routing table and on the operating system (PVSS runs faster on Linux than Windows).

References

[1] Alfred V.AHO, Jeffrey D.Ullman, Foundations of the Computer Science. C Edition: Computer Science Press. An imprint of W.H. Freeman and Company. New York. 1995. ISBN 0-7167-8284-7. 786 p.

[2] Thomas Kyte, Effective Oracle by Design, 2003. Ed Osborne: Oracle Press. ISBN 0-07-223065-7. 646 p.

[3] DIM Distribution Interface Management, http://dim.web.cern.ch/dim/

[4] CORBA, http://www.corba.org/

Chapter 10 Validation of the CIC_DB_lib

This chapter describes the different tests carried out to validate the object layer, namely the CIC_DB_lib (C code) and its two bindings (PVSS and python) and the GUI layer too.

The first part describes common tests to verify that the functions of the CIC_DB_lib work correctly. It includes tests on the bulk collect insert and update and on the automated updates further to a change in the connectivity or an update of the status of a device. The second part shows how the HCAL connectivity has been inserted using CIC_DB_lib and how modules could be configured using the PVSS CIC_DB_lib to get connectivity information. The third part explains how a slice of the VELO connectivity has been inserted using the Python CIC_DB_lib. The VELO connectivity includes the microscopic level too. So we could validate the functions related to inserting and querying microscopic view. Finally I present some tests to simulate the device history. In the tests, updating wrongly a status of devices was performed such as updating the status of a destroyed device.

10.1 Validation of the insert and update statements

10.1.1 Test Frame

The different tests have been carried out using C, python and PVSS programs. A C program has been implemented to verify the behavior of queries when run concurrently.

Some of the scripts used for tests are stored on the CERN network at: dfs (G:\Experiments\Lhcb\group\TFC\CICDBproject).

The main points I wanted to check were:

- If the functions built with their interfaces (Python + PVSS) are doing what they are supposed to do.
- The behavior of the functions (especially update, delete and insert) in case of user errors or constraint violations.
- The behavior of some functions (insert and update functions) when run concurrently.
- The behavior of functions when performing bulk inserts or updates.
- The automatic updates of information related to paths when there is a change in the connectivity table (insert, delete and update a link).

Finally the CIC_DB_lib and its interfaces have been validated by their use in different projects (HCAL, VELO, DAQ and TFC).

10.1.2 Multiple insertions

The TFC and DAQ connectivity presented in Chapter 5, has been inserted using functions included in CIC_DB_lib. I have written a C application for each subsystem. The following functions have been used:

- **InsertMultipleDeviceTypes** to insert many device types in one go;

- **InsertMultipleFunctionalDevices** to insert many functional devices;
- **InsertMultiplePorts** to insert many ports;
- **InsertMultipleSimpleLinkTypes** to insert link types;
- **InsertMultipleMacroLinks** to insert links between devices.

A simple way to verify that the insertions were correct in terms of number of rows and right data included in the right place (included NULL values) was to query them using functions get information about a device type row, a device row, a port row, a link type row and link row. The same type of tests was performed for the update and delete based functions. So with this way of doing, the user can check that what he has inserted was what he wanted.

I also faked some errors such as links starting from an already used **portid** or ports belonging to a non-existent functional device. These tests were meant to verify the database constraints and the error handling.

10.1.3 Memory leak

The CIC_DB_lib includes functions such as getting the paths between two devices or inserting many rows in one go (the initialization of the cache was properly done), which perform a lot of memory allocation. Using the Valgrind tool [1], one could verify if there are memory leaks and solve the problem. In my case, I could find some blocks that were not released.

The methodology used was to write an executable which calls these functions and then call
`valgrind with –tool=memcheck --leak-check=yes <name of the executable>` to detect memory leaks.

10.1.4 Verification of the autonomics features

Some of the functions to update information have been tested when creation the dhcp config file (Chapter 9). Nodes and links have been excluded.

I have also verified that after:

- updating **nodeused**, or after updating a link attribute such as **bidirectional_link_used, linktypeid, lkused, system_name** part of the TFC or DAQ, the updates of PATH_LINES, ROUTING_TABLE and DESTINATION_TABLE were performed.
- deleting of a device, of a port and of a link which happens in the DAQ or TFC system, the PATH_LINES, ROUTING_TABLE and DESTINATION_TABLE were updated dynamically;
- inserting a link, PATH_LINES, ROUTING_TABLE and DESTINATION_TABLE were updated dynamically;
- inserting or changing the status of a device was automatically reported in the DEVICE_HISTORY (including the components of a board if any);
- changing the status of a device was performed in a coherent manner (the required updates to other tables were made, such as updating the status of the board components if necessary);
- swapping two devices was allowed (same type and same connectivity).

For inventory/history information updates and deletions, giving incoherent input parameters have been tested to verify that the changes were not performed and nothing was blocked.

10.1.5 CDBVis

CDBVis is another way to validate CIC_DB_lib as it uses insert and update based functions. It permits to verify that all the links have been inserted. Referring to Figure 123, the output connectivity of the VELO_REPEATER_BOARD_00 has not been inserted yet as the last type of devices in any subsystem dataflow is the TELL1 boards.

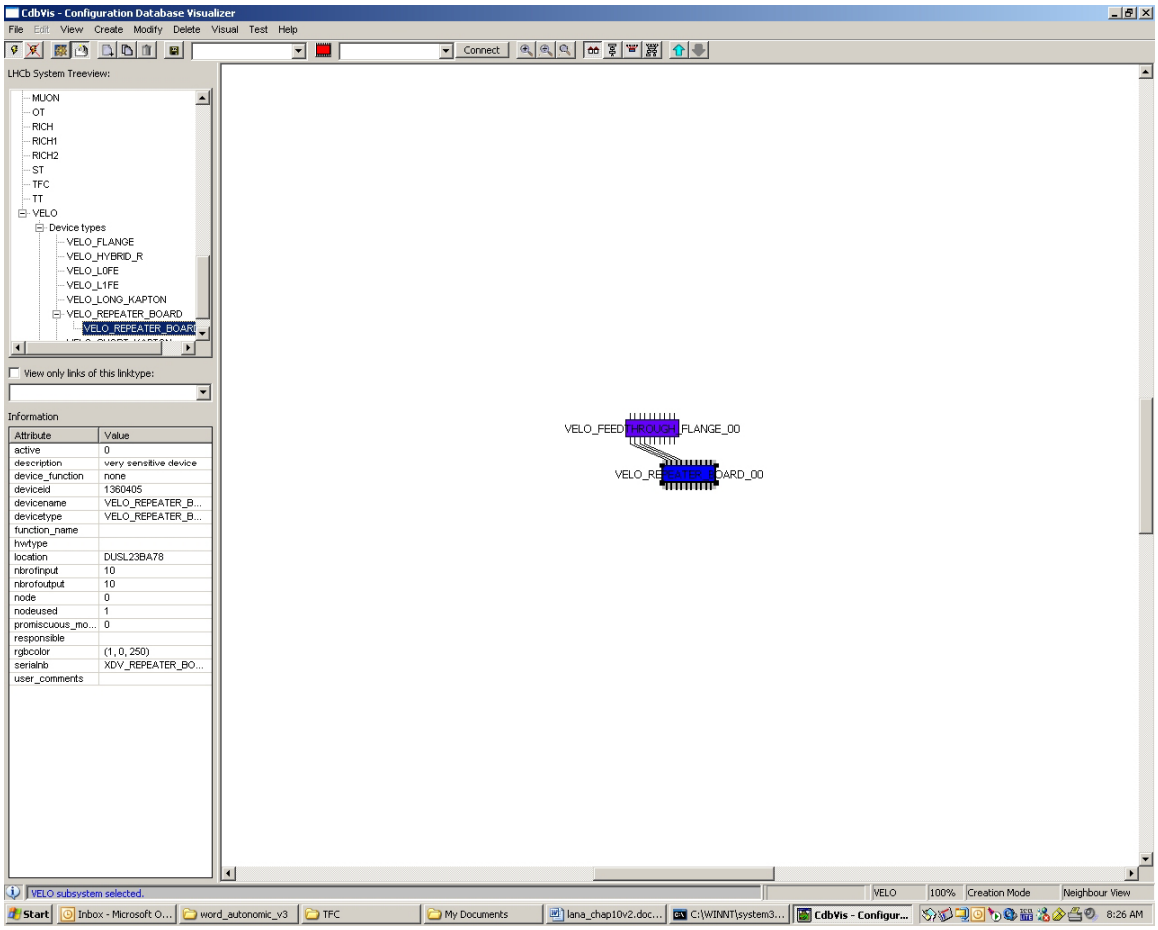


Figure 123. Example of an incomplete connectivity for the VELO_REPEATER_BOARD_00.

Part of the MUON connectivity has been inserted in the CIC DB using CDBVis. On the opposite, it was a good way to test and debug CDBVis too. For instance, we found bugs when viewing paths (not the correct last node).

10.2 Use of CIC_DB_lib and its PVSS binding by the CALO subdetector

In Chapter 2, section 2.1.3, there was a need to get the connectivity between devices to configure the modules. The connectivity of the HCAL has been described in Chapter 2, section 2.1.3. The next two subsections explain the use of CIC_DB_lib to insert and query connectivity information.

10.2.1 Inserting the connectivity in the CIC DB

Configuration information is used to get the SPECs addresses of the hardware and connectivity information will give the DAC board name, INT board name and the FE name which drive the given channel name (not direct connection).

Text files exist which include device types, devices and links between devices. Thus I could insert the connectivity with CIC_DB_lib. Around 14,000 links were inserted. There are 1488 channels, 1488 PMTs, 52 LED1s, 52 LED2s, 8 DACs, 4 INTs, 4 FEs and 4 Controls PCs. (see Appendix L for the C code written to insert the HCAL connectivity).

Inserting the connectivity is done as follows (order to respect the database constraints):

1. Insert all the device types of the system (HCAL_CHANNEL for instance);
2. Insert all the functional devices with their serial number (HCAL_CHANNEL_001);
3. Insert all the ports group by functional devices;
4. Insert all the link types (data_signal);
5. Insert all the links between (functional device, port number).

The insertion was successful as we check that the number of devices sorted by types, the number per ports and links were the same as the numbers in the text files.

However it is up to the user to ensure that all the devices, ports and links have been inserted. There is no way to know in advance how many devices should be inserted per subsystem for instance.

10.2.2 Getting the connectivity between 2 devices

The CALO group uses the PVSS binding of CIC_DB_lib.

Use case 1 (Chapter 2, section 2.1.3) can be solved by getting the paths between a channel and a DAC, a channel and an INT and finally between a channel and a FE board. In Use case 2 and use case 3, the connection involved is point-to-point connection as a channel is directly linked to two LEDs and to a PMT. Their requirement was to get the connectivity between all their channels and DAC, INT and FE in less than 100 s.

To respond to their requirements, I suggested them to use **PVSSGetDetailedConnBetweenDeviceDevType.** This function allows getting detailed connectivity between a given device and a device type.

Example of usage (PVSS script):

```
dyn_string nfrom_list, pfrom_list, nto_list, pto_list, lkinfo_list,
devicename_list;
dyn_int         pwayfrom_list,         pwayto_list,         pid_list,
lkpos_list,deviceid_list;
dummy=PVSSDBConnexion(dbname,login,passwd,errmess);
//Get all devices of type HCAL_DAC
dummy=PVSSGetDeviceNamesPerType("HCAL_DAC",devicename_list,
deviceid_list);
if(dummy==0)
{
t1=getCurrentTime();
for(i=1;i<=dynlen(devicename_list);i++);
```

```
{
devicename_ch=devicename_list[i];
            if(i==1)
            {
      //Get the connectivity between a given HCAL_DAC and channels
      dummy=PVSSGetDetailedConnBetweenDeviceDevType(devicename_ch,"HC
AL_CHANNEL",1,nfrom_list,    pfrom_list,    pwayfrom_list,    nto_list,
pto_list,    pwayto_list,    pid_list,lkpos_list,    lkinfo_list,1,0,
errmess);
            }
            else
            {
                  if(i==dynlen(devicename_list))

      dummy=PVSSGetDetailedConnBetweenDeviceDevType(devicename_ch,"HC
AL_CHANNEL",1,nfrom_list,    pfrom_list,    pwayfrom_list,    nto_list,
pto_list,    pwayto_list,    pid_list,lkpos_list,    lkinfo_list,0,1,
errmess);
                  else
      dummy=PVSSGetDetailedConnBetweenDeviceDevType(devicename_ch,"HC
AL_CHANNEL",1,nfrom_list,    pfrom_list,    pwayfrom_list,    nto_list,
pto_list,    pwayto_list,    pid_list,lkpos_list,    lkinfo_list,1,0,
errmess);
            }
}
}
```

10.2.3 Verification of the execution time requirement

This script has been executed on a Windows machine and on a Linux machine. It returns the detailed paths between each DAC and a CHANNEL. The Linux and Microsoft Windows Server 2003 machines have similar characteristics which are Intel Xeon 2.8 GHz and 2 GB of memory.

Try	Execution time (s) C code Windows	Execution time (s) C code Linux	Execution time (s) PVSS code Windows	Execution time (s) PVSS code Linux
1st try	5.29/6.02	4.87/5.38	6.62	6.35
2nd try	4.45/5.18	4.45/4.96	7.52	5.29
3rd try	4.44/5.19	4.32/4.83	6.17	5.33
4th try	4.44/5.17	4.30 /4.81	6.58	5.10
5th try	4.5/5.23	4.38/4.81	6.12	5.07
Avg	4.62/5.36	4.46/4.95	6.60	5.42

Table 27. Execution time of the script.

In Linux, the C code is executed faster than in Windows (a few ms faster).

It is because PVSS is faster on Linux. In both cases, the first call to **GetDetailedConnBetweenDeviceDevType** consumes 90% of the execution time in Linux and 86.2% in Windows. This is because the first call loads the connectivity table of HCAL in memory (roughly 14,000 links). In the query, there is a union statement to revert bidirectional links. And the select query itself involves 3 joins (**FUNCTIONAL_DEVICES**, **CONNECTIVITY** and **PORT_PROPERTIES**

tables). The other calls do not perform this operation as the connectivity table (of the HCAL is already loaded into memory).

So it depends on two factors, the load on the database and the load on the network.
The database (Oracle 10g) is a central one accessed by hundreds of users which can run heavy processes. The load on the database is already quite heavy. The result of the tests was more or less the same (the worst result I got was 20 sec which is still less than 100 sec). However it is important to note that the CIC DB will be installed in the pit and accessed only by the LHCb group.

The PVSS script is also executed faster in Linux than in Windows.

However the requirement is satisfied with the current performance (it is far beyond the 100 sec limit). Thus the functions which get the path between two modules and between a module and a type of module could be validated.

10.3 Inserting and querying the VELO connectivity

In Chapter 2, in section 2.5.2.2, a slice of the VELO connectivity from a hybrid to a TELL1 board has been presented. Each hybrid has the same connectivity schema. A hybrid is connected to four short kaptons (similar to cables). A short kapton is connected to a long kapton which is connected to a port of the feedthrough flange (similar to a patch panel). A port of this device is connected to a port of a repeater board via interconnects (also like cables). This repeater is connected to one TELL1 board, to a control board and a temperature board. A control board drives 6 hybrids and a temperature board, 16.

10.3.1 Using the connectivity for debugging purposes

The VELO group wants to save the connectivity for debugging and management purposes. If the long kapton VELO_LGKAPTON_00 (for instance) fails, they want to know all the devices affected by it.

Unlike other subdetectors, they want to know which beetles (silicon chips located on the hybrid) are associated to a given driver mezzanine (which sits on a repeater board).

So there is a need to describe the internal connectivity of the hybrid and the repeater boards as explained in Chapter 2. The internal connectivity of the feedthrough flange has also been stored as mentioned in Chapter 2.

10.3.2 Inserting the macroscopic and microscopic connectivity

The connectivity of the VELO will be inserted into two steps. The first step is to insert the macroscopic connectivity from the hybrid to the repeater board. The same functions have been used as for the HCAL.

The second step is to insert the internal connectivity of boards (hybrids, repeater boards and feedthrough flanges). The order of inserting the microscopic connectivity is similar to the macroscopic one. The only difference is there is no need to insert the ports of a microscopic device. The Python code (written by the VELO group based on my advices) below shows an example how to insert the 4 driver mezzanine cards of the repeater boards. It also shows how to insert the micro links of these 4 driver mezzanine cards.

```
#microscopic devices making up one 'slice' of the VELO


#DRIVER_MEZZANINE (inserting the 4 driver mezzanine cards of the
#repeater board)
cfDB.InsertMultipleBoardCpnts('VELO_DRIVER_MEZZANINE_0','VELO_DRIVER_
MEZZANINE',1,'VELO_REPEATER_0','4TVLAURPTA0010','','rshade','0_TOP_LE
FT_0_J3','',1,0)
cfDB.InsertMultipleBoardCpnts('VELO_DRIVER_MEZZANINE_1','VELO_DRIVER_
MEZZANINE',1,'VELO_REPEATER_0','4TVLAURPTA0011','','rshade','0_TOP_LE
FT_0_J5','',0,0)
cfDB.InsertMultipleBoardCpnts('VELO_DRIVER_MEZZANINE_2','VELO_DRIVER_
MEZZANINE',1,'VELO_REPEATER_0','4TVLAURPTA0012','','rshade','0_TOP_LE
FT_0_J7','',0,0)
cfDB.InsertMultipleBoardCpnts('VELO_DRIVER_MEZZANINE_3','VELO_DRIVER_
MEZZANINE',1,'VELO_REPEATER_0','4TVLAURPTA0013','','rshade','0_TOP_LE
FT_0_J9','',0,1)

#Get the deviceid of VELO_REPEATER_TEST, a macroscopic component
devid=cfDB.GetDeviceID_devicename("VELO_REPEATER_0");

#Get the portid which corresponds to (deviceid,port_nbr,
#port_type,port_way) There is a bijection between portid and these
#parameters

#From REPEATER input to MEZZANINE (insert the micro links between
#repeater input and mezzanine)
portid=cfDB.GetPortID_portinfo(devid,"0","data",1);
cfDB.InsertMultipleMicroLinks('motherboard','VELO_DRIVER_MEZZANINE_0'
,portid,0,'mixed_data',0,1,0)
portid=cfDB.GetPortID_portinfo(devid,"1","data",1);
cfDB.InsertMultipleMicroLinks('motherboard','VELO_DRIVER_MEZZANINE_1'
,portid,0,'mixed_data',0,0,0)
portid=cfDB.GetPortID_portinfo(devid,"2","data",1);
cfDB.InsertMultipleMicroLinks('motherboard','VELO_DRIVER_MEZZANINE_2'
,portid,0,'mixed_data',0,0,0)
portid=cfDB.GetPortID_portinfo(devid,"3","data",1);
cfDB.InsertMultipleMicroLinks('motherboard','VELO_DRIVER_MEZZANINE_3'
,portid,0,'mixed_data',0,0,1)

#From MEZZANINE to REPEATER output (insert the micro links between #
#mezzanine and output repeater)

portid=cfDB.GetPortID_portinfo(devid,"0","data",2);
cfDB.InsertMultipleMicroLinks('VELO_DRIVER_MEZZANINE_0','motherboard'
,0,portid,'mixed_data',0,1,0)
portid=cfDB.GetPortID_portinfo(devid,"1","data",2);
cfDB.InsertMultipleMicroLinks('VELO_DRIVER_MEZZANINE_1','motherboard'
,0,portid,'mixed_data',0,0,0)
portid=cfDB.GetPortID_portinfo(devid,"2","data",2);
cfDB.InsertMultipleMicroLinks('VELO_DRIVER_MEZZANINE_2','motherboard'
,0,portid,'mixed_data',0,0,0)
portid=cfDB.GetPortID_portinfo(devid,"3","data",2);
cfDB.InsertMultipleMicroLinks('VELO_DRIVER_MEZZANINE_3','motherboard'
,0,portid,'mixed_data',0,0,1).
```

This python code below shows how to insert the internal connectivity of the feedthrough flanfge.

```
#FEEDTHROUGH_FLANGE(insert the internal connectivity of the
#feedthroughflange)
```

```
for i in range (1,20):
      devid                                                              =
cfDB.GetDeviceID_devicename("VELO_FEEDTHROUGH_FLANGE_0");
      portid1 = cfDB.GetPortID_portinfo(devid,"%s" % i,"data",1);
      portid2 = cfDB.GetPortID_portinfo(devid,"%s" % i,"data",2);
      cfDB.InsertMultipleMicroLinks('motherboard','motherboard',porti
d1,portid2,'mixed_data',0,1,1)
```

10.3.3 Getting the connectivity between VELO devices

The same set of functions is used to query paths between devices as in the HCAL such as `GetDetailedConnectivityBetweenDevices` which returns the detailed paths between 2 devices.

To get the 4 possible paths (and not 16) between a hybrid and a repeater board, the algorithm to get the paths (the same as used in the HCAL but the input parameters are different) checks if the node to be added in the current path has an internal connectivity. If yes, it checks if a signal arriving at a given input can go out from the given output using `CheckInternalConnectivity`. This function returns 0 (for OK) and -1 for (not OK) given two portids (input and output of the same device). For instance, the input port 1 of the feedthrough flange is not compatible with the outport 2, the function returns -1.

If not, it considers any combination of (input, output).

The VELO connectivity allows testing the functions related to microscopic devices and connectivity.

10.4 Simulation of device history

10.4.1 Introduction

The real inventory information has not been inserted into the CIC DB so far, since the history of a device begins with the start of the LHC.

However in CIC_DB_lib, there is a set of functions which enable to:

- Update the status of a hardware device or a functional device;
- Get the history of a hardware or functional device, filtered by date;
- Get the current status of a hardware or functional device;
- Get all the functional or hardware devices which are in a given status filtered by subsystem.

These functions are included in the PVSS binding but not in Python as these functions should be used from PVSS. It is part of the hardware monitoring.

The date format is the same one used in all the functions and is equal to YY/MM/DD/HH24/MI/SS.

10.4.2 Test patterns

Each function related to inventory information has been tested individually using the use cases defined in Chapter 4, section 4.3.4.

The following tests have been performed:

- Update the status of a hardware device IN_USE to SPARE with a replacement and with no replacement;
- Update the status of a hardware device IN_USE to TEST with a replacement and with no replacement;
- Update the status of a hardware device EXT_USE to IN_USE;
- Update the status of a hardware device IN_REPAIR to DESTROYED;

Also some impossible patterns have been tested to verify that no update was done:

- Update the status of hardware device DESTROYED to SPARE;
- Update the status of a hardware device IN_USE to SPARE with a replacing hardware IN_USE;
- Update the status of a hardware device IN_USE to TEST with a test board not free;
- Update the status of a hardware device EXT_USE to IN_USE, and the functional device is already IN_USE.

Functions based on select have also been tested.

Example of code:

```
//Get the status of functional device "TEST_BOARD_1"
res2=GetFunctionalDeviceStatus("TEST_BOARD_1",resultList ,ErrMess);
//Get the status of hw device "CC21PP78"
res2=GetHWDeviceStatus("CC21PP78",resultList , ErrMess);
//Replace the hw device which occupies the functional device
//"ttcrx_1000" with "CC21PP78" and set the status of the replaced
//device to SPARE
res2=ReplaceFunctionalDevice("ttcrx_1000","SPARE","Liverpool_Uni","no
ne","12/55/41/02/05/06","CC22PP78","12/55/41/02/06/06",ErrMess);
//set the status of the hw device which occupies the functional
device //"TELL1_Board_77" to EXT_USE
res2=ReplaceFunctionalDevice
("TELL1_Board_77","EXT_USE","Liverpool_Uni","device in test
","11/08/45/10/02/06","none","none", ErrMess);
//set the status of the hw device which occupies the functional
device //"TELL1_Board_12" to TEST and replace it with "CC21PP78"
res2=SetToTestUseStatus("TELL1_Board_12","none","06/04/10/12/24/25","
CC21PP78","TEST_BOARD_1","06/05/22/12/05/06", ErrMess);
//set the status of the hw device "88XX745P45SS" to IN_USE and it
occupies "TELL1_Board_77"
res2=UpdateHWDeviceStatus("88XX745P45SS","IN_USE","none","getting
back the hw","12/55/41/02/06/06","TELL1_Board_77",ErrMess);
// set the status of hw device "XX3356UGD" to IN_USE and occupies the
//functional device "ttcrx_1000". This one does not work as
"ttcrx_1000" //is already IN_USE
res2=UpdateHWDeviceStatus("XX33356UGD","IN_USE","none","getting back
the hw","12/55/41/02/06/06","ttcrx_1000",ErrMess);

res2=GetHistoryOfHWDevice("XX33356UGD",ipaddList,len_array,"none","no
ne",ErrMess);
```

10.5 Validating the connectivity information

The different functions related to query connectivity information have been tested through different means.

- CDBVis uses functions which give neighborhood connectivity of a device (inputs and outputs) and give all the paths through a device;
- TFC Local control uses functions which return neighborhood connectivity of a device and which return the output port number given a subsystem;
- The CALO project uses functions which return the detailed connectivity between a device and a device type or between two devices.
- The VELO project uses functions to insert the internal connectivity of boards.

10.6 Conclusion

In this chapter, we have described different tests which have been performed to verify and validate the functions of CIC_DB_lib and its two bindings. The tests covered insertion, update and deletion and connectivity and inventory/history information retrieval. There were also some simulations of mistyping in the input parameters or bad behavior of the user as inserting twice a same device, updating the status of a device which is destroyed, swapping two devices which are of different types. The main purpose was to predict the robustness of CIC_DB_lib in case of bad usage.

References

[1] Valgrind, http://valgrind.org/

Chapter 11 Other examples where the CIC DB can be implemented

This chapter describes other areas than LHCb where databases are used to configure experiments. The first section is devoted to configuring telescopes. There are some common needs which can be modeled with the CIC DB. The second section presents another HEP experiment, ATLAS in which an Oracle relational database has been used to configure the MUON detector. We show that the CIC DB design and implementation could have been applied in this context.

11.1 Configuring telescopes

Large telescopes such the ones used in the VLT (Very Large Telescope) [1] or ALMA [2] (Atacama Large Millimeter Array) project need to be configured.

In both projects, they have implemented their own DBMS to configure their equipment. The next subsections aim at giving the outlines of their software architecture.

11.1.1 VLT project

The VLT is composed of an array of four identical main telescopes. Each of them has a mirror of 8 m diameter. They can be used independently or as a single instrument. The first telescope has been installed in 1996. The control software for the VLT is a distributed system of 120-150 coordinating workstations and microprocessors based on VME systems (Local Control Units, LCUs). The LCUs control the mechanical, electrical and optical components of the VLT in real time. The synchronization of the computing units is either based on hardware (faster as it does not go through all the OSI layers) or on software (via LANs).

The VLT architecture includes an online database which provides real-time data capabilities. This database is a home-made product. The main reason is at that time, the performance of writing into a database was too low. So this database can be viewed as a set of files.

Each LCU loads the information it needs to configure and control its equipment from a central database. Each local database has a hierarchical structure which reflects the physical objects that constitute the telescope system (instruments, telescope control, components …). It also describes how these components are logically contained, in term of “part-of”. So the On-Line database contains information about component configuration and also how components are nested (hierarchical relationship). There is no information related to the connectivity as in their case, the connectivity is trivial. Also for this project, they did not plan to store history information about components.

The database implementation is based on Object Oriented terminology. They have a lot of classes such as motors, encoders as shown in Figure 124. Each class has many instances.

They have implemented tools to interact with the database. Templates are widely used to describe the different modules. They have introduced a simple description language for the database structure to handle the definition of instances, to define a new class, to take proper default values if missing.

A telescope description comprises 6400 instances, i.e. 6400 entries in the database.

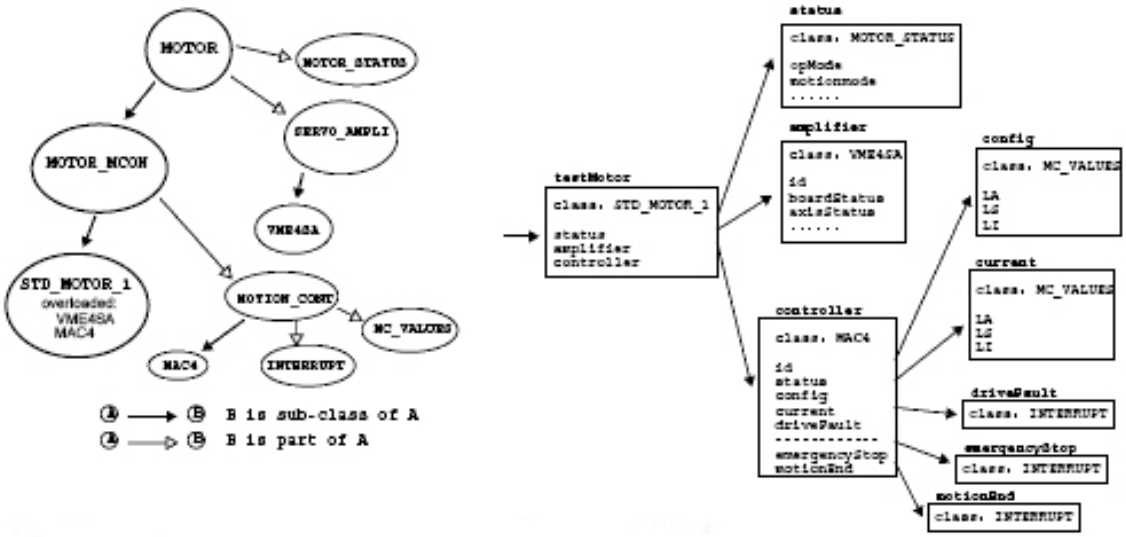


Figure 124. Example of relations between classes.

11.1.2 ALMA project

The ALMA telescope consists of 64 12m diameter antennas which operate in the millimetre and sub millimetre wavelength.

They have developed a framework called ACS (ALMA Computing Software) based on CORBA. This framework has been embedded in languages such as JAVA and Python. The objective of this framework is to provide a set of tools to build and monitor the system.

ALMA is composed of a complex collection of hardware and software. They need to maintain information about the current configuration in use, the previous configuration to allow reverting back in case of problems and the next planned configuration (which is in development).

The telescope configuration changes with time, for instance if a device fails or if an antenna or weather station is taken offline. According to the cause of change, it can imply a change in the hardware or/and software configuration.

The detailed telescope configuration is tracked in the telescope configuration database. This database is integrated in the control system. At start up, the control system uses the Telescope configuration database to know what devices should be online.

When an antenna is placed online, it has to declare itself by giving the serial numbers of its parts.

This information is then compared with the content of the Telescope Configuration database. If there is mismatch of the information, the database can be updated by inserting this new information automatically or it can cause an error in the initialization of the antenna.

So the Telescope Configuration database corresponds to a snapshot of the telescope at a given time, in fact it changes whenever there is change in the telescope configuration.

Unlike the previous project, history information is stored since a device failure causes a change in the configuration.

They use an Oracle relational database to store the information and Java database functions to query information. They also use XML packages provided by Oracle (XPath and XQuery) to describe their devices and their software. Figure 125 [3] shows in detail how the database schema looks like. There are 3 main concepts,

- hardware devices (in light yellow) which are identified by a serial number. Properties with their values are also described here;
- configuration information which describes the previous, the current, and the next configuration used and to be used, and also all the devices which are part of this configuration (represented in green);
- software configuration represented in blue.

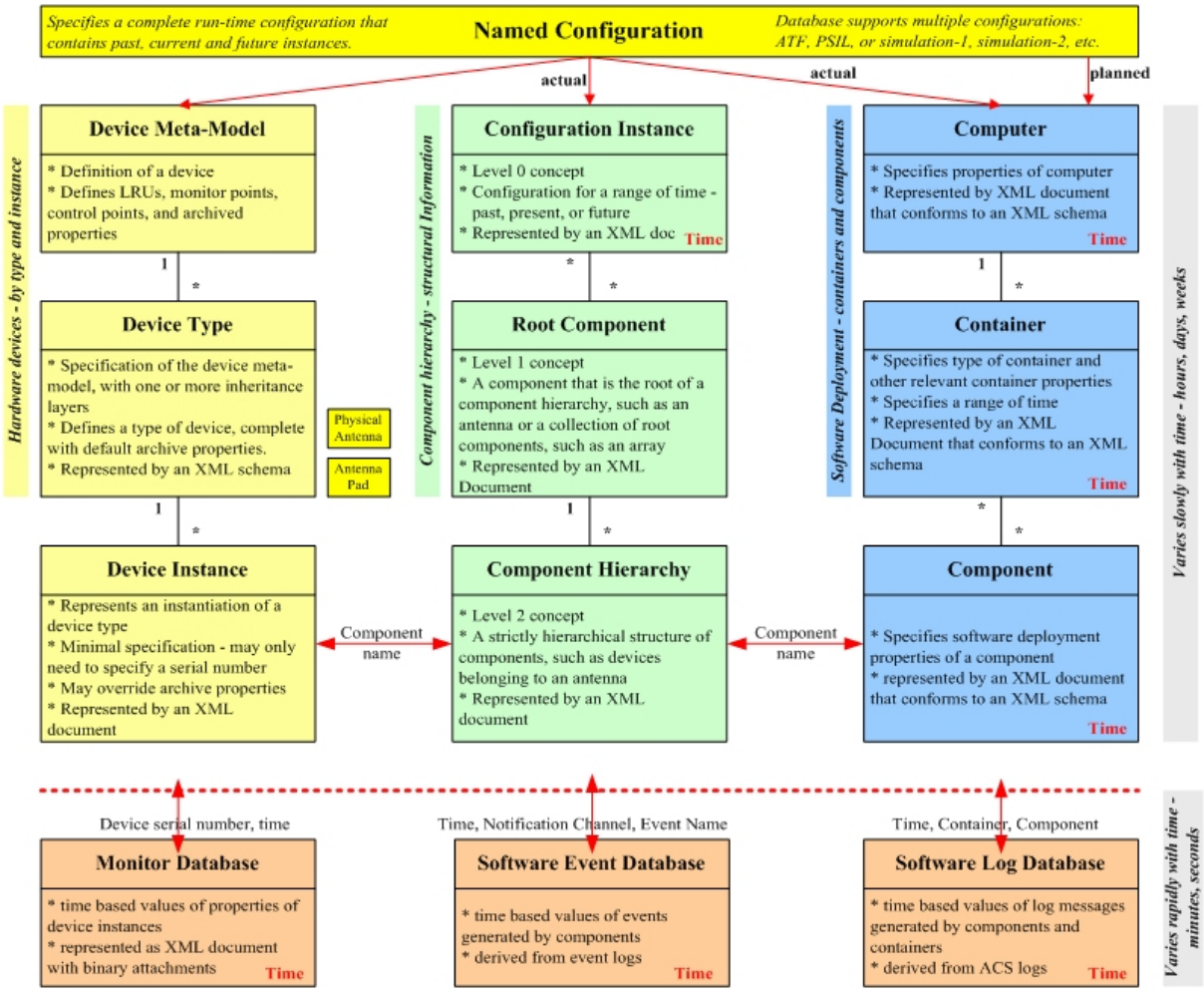


Figure 125. ALMA control system architecture.

On the top of the red dashed line, elements vary slowly with time (hours, days, weeks). On the bottom of the red dashed line, elements vary rapidly with time (minutes, seconds).

Red arrows represent the interactions between the three concepts. The **Named Configuration** (in dark yellow) consists of hardware devices, configuration and software information.

The red arrows on the bottom specify which parameters are required by the three databases whose data varies quickly with time.

11.1.3 Use of the CIC DB in the VLT or ALMA project

Using documents on the VLT and ALMA projects and discussions with a person involved in these two projects, part of the CIC DB can be implemented in the ALMA project (the VLT project is old). The connectivity information in this context is not needed as there is only one system with few links. The configuration and history design can be applied in their context with some changes. They do not have the concept of hardware/functional duality as they only use serial numbers to identify the equipment. So the configuration is related to the hardware identifier and not to the functional device. History information (in ALMA project) consists of only tracking hardware information.

To make it compatible with the schema of the CIC DB, the functional device can be set equal to the hardware serial number.

11.2 Application of the model to the thin Gas Chamber of ATLAS

11.2.1 Description of the problem

The ATLAS MUON group has developed the Thin Gas Chambers database [4]. Thin gas chambers (TGCs) are used by the MUON trigger system in the forward region of the ATLAS experiment. They have designed a relational database based on Oracle Technology to configure their system.

The database will contain configuration parameters of equipment as in the CIC DB. It will also describe the structure of the system in terms of hierarchies. Using the information stored in this database, it should be possible to retrieve from which chamber, a given electronic module gets its signal and to which trigger coincidences it contributes. In their case, they have several different hierarchies (DAQ, Trigger, etc.). In parallel, they want to navigate from one hierarchy to another one.

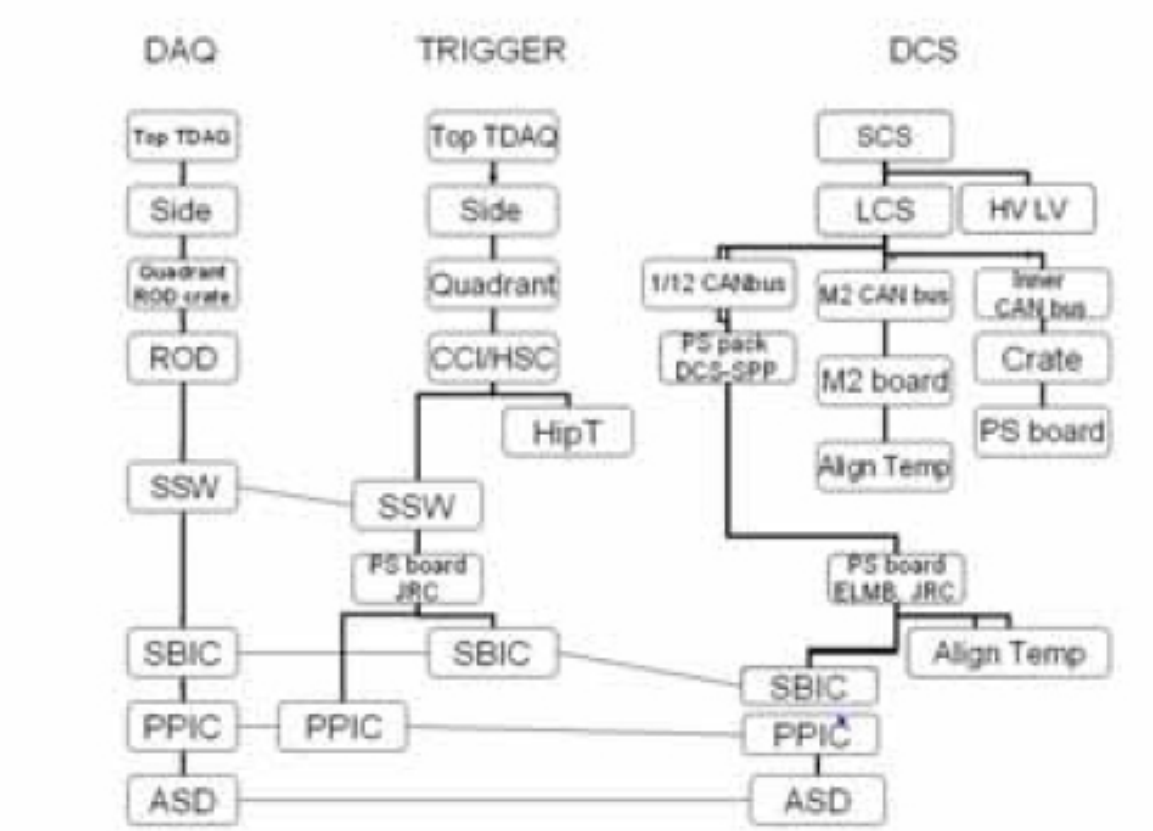


Figure 126. Structure of the thin gas chambers.

Their table design is different from the CIC DB table schema. I would like to show that the schema of the CIC DB can also be applied in the context of the ATLAS thin gas chambers.

11.2.2 ATLAS database design

Figure 127 shows how ATLAS thin gas chamber database has been designed.

- DEVICE table contains the general detector elements such as ASIC, CHAMBER. Each of them is uniquely identified by *oid***.**
- ARGUMENT stores parameters related to the previous elements via ARGUMENT.*oid*. ARGUMENT.*dev_id* is a foreign key to DEVIC*E.oid*. The value of this parameter is not stored in this table. This table defines the structure of a DEVICE.
- PRODUC*T* contains instances of elements in DEVICE table, uniquely identified by PRODUCT.*oid*. PRODUCT.*devid* is a foreign key to DEVICE.*oid*.
- CONFDB contains the values of a particular configuration identified by CONFDB.*conf_id.* CONFDB.*oid* is a foreign key to ARGUMENT.*oid* and CONFDB.*prod_id* is a foreign key PRODUCT.*oid.*
- CONFIGURATION contains all the generic information of a configuration such as name, date of creation, etc.
- HIERARCHY contains the link between devices. There is also a field called hierarchy which corresponds to the type of the hierarchy. It is not represented in

Figure 127. With this table, they also model connectivity between devices and not only hierarchy.

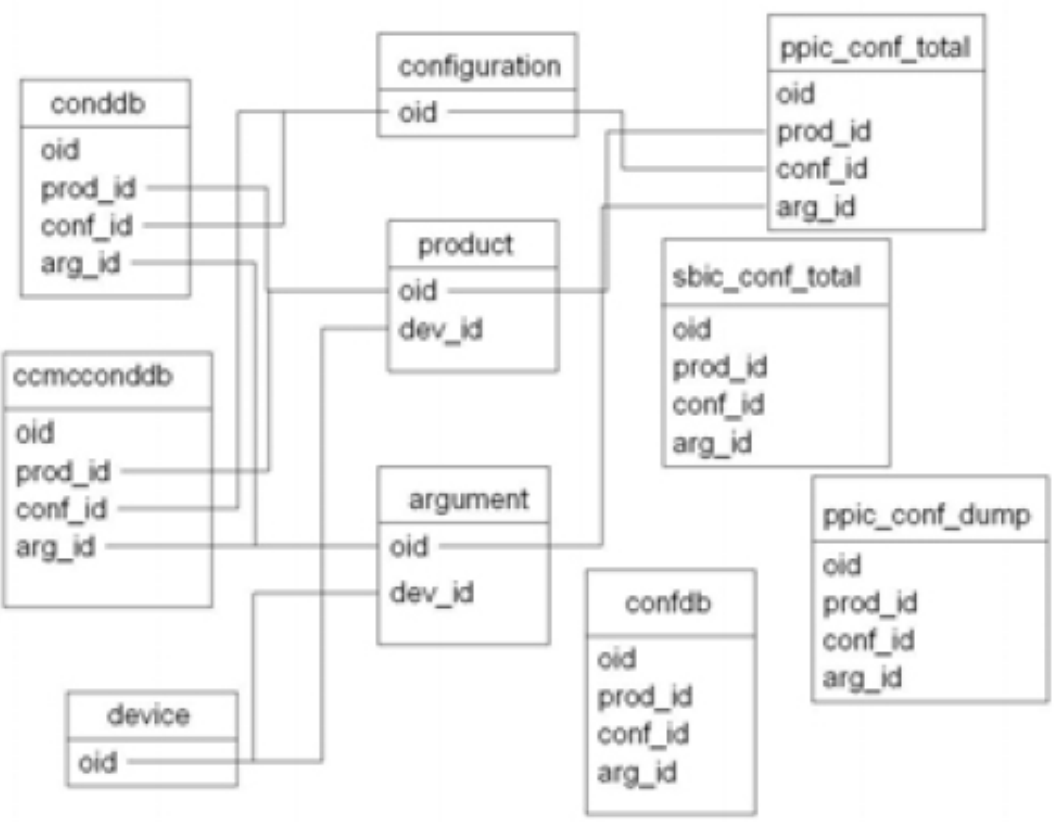


Figure 127. Design of ATLAS database.

11.2.3 Use of the CIC DB in ATLAS MUON Chamber database

The CIC DB table design is slightly different, especially regarding recipes. In the table schema for the recipes presented in Chapter 5, there is nothing similar to the ARGUMENT table, i.e. there is no table which contains the structure of a device type. They just save the attributes with its values part of the recipe. The Table 28 shows the mapping between the ATLAS MUON Gas Chamber database schema and the CIC DB schema. So for instance, the information in the ARGUMENT table can be retrieved using RECIPE_DATA(id, propname, propid).

Moreover in JCOP design, a recipe is linked to a HIERARCHY which is not the case in the ATLAS table design.

Regarding the HIERARCHY table in ATLAS design, the CONNECTIVITY table can store links between devices. In my case, the granularity is even deeper as I store it at the port level. So in their case, they just create one port per device. They can put the hierarchy type in the *link type* or even in the *system_name*.

Also for hierarchy data, they can store it in the HIERARCHY table part of JCOP or also in CONNECTIVITY table as follows. They create if needed a port for the connectivity and as many as needed for the hierarchy. In the first case, they set *port_type* to “connectivity” and in the second case they set *port_type* to “hierarchy”.

However it is important to note that LHCb table design can be applied to the ATLAS database design but the opposite is not true as they do not have a port concept.

Table in ATLAS Design	**Tables in the CIC DB**
DEVICE	FUNCTIONAL_DEVICE_TYPES
CONFDB	RECIPE_DATA(propid, propvalue)
ARGUMENT	RECIPE_DATA(id, propname, propid)
PRODUCT	FUNCTIONAL_DEVICES
HIERARCHY	ITEMS, HIERARCHIES, CONNECTIVITY
CONFIGURATION	RECIPES, RECIPE_TAGS

Table 28. Applying the CIC DB schema to the ATLAS Thin gas chamber.

References

[1] VLT project,
http://www.eso.org/projects/vlt/sw-dev/wwwdoc/APR2004/dockit.html

[2] ALMA project,
http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ConfigurationDatabase.pdf

[3] A Redesign for the ALMA Monitoring Database, Allen Farris, April 2006, http://almasw.hq.eso.org/almasw/pub/CONTROL/MonitorDatabase/RedesignV2.ppt

[4] Y.Benhammou, S.Bressler, E.Etzion, D.Lellouch, L.Levinson, S.Tarem, The Thin Gas Chamber Database preparations for ATLAS. In the Proceedings of 14th Conference on real time (RT 2005) Stockholm, Sweden, June 4-10, 2005.

Conclusion

Telle est la vie
Tomber sept fois
Et se relever huit.
Poème populaire japonais.

This thesis describes the methodology applied to build an autonomic LHCb ECS with a focus on the experiment configuration. In other words, it describes how the ECS can configure the LHCb experiment by a single-click.

There are often changes in the technology used to build the LHCb experiment. This thesis has described the LHCb experiment as expected to be built in August, 2006.

Due to the complexity introduced by the scale of the experiment with several hundred thousand of electronics modules to configure in different ways, an autonomic architecture was required by the Experiment Control System. The software architecture which has been implemented to handle the configuration is a 3-Tier architecture. It consists of a database layer (CIC DB and PL/SQL applications), an object layer (Two Perl scripts, CIC_DB_lib and its two bindings and the PVSS library for recipes) and the GUI layer composed of CDBVis and PVSS panels. The object layer has been developed in such way that autonomics tools for the configuration of the experiment can be built on top of it.

The CIC DB is a relational database, implemented using Oracle technology. It contains the different recipes of devices, the connectivity between devices for the whole experiment and inventory/history information of a device. To build the schema, we have first identified the different groups of users of the CIC DB. For each group, their needs in terms of configuration, inventory/history and connectivity have been defined.

Based on these use cases, we have proposed a generic model for connectivity and inventory/history. The main advantage of a generic schema is an easier management of the database. Despite the complex environment (many devices with different technologies and different types of topologies), the table schema is rather simple. It is readable and easy to extend. It can also be used by other experiments. ITER have expressed an interest in our design. The inventory/history use case consists of providing information about the history of a device (its different statuses over years) and also the current statuses of devices (spare devices, broken devices, devices in use, etc.). It introduces the concept of duality between hardware and functional devices. In other words, a functional device can be viewed as an entity which fulfills specific tasks whereas a hardware device is the physical equipment. A functional device is occupied by a hardware device. Both hardware and functional devices have a status and a history. The possible statuses of a hardware device are different from the functional device. A hardware device can have status “DESTROYED” whereas for functional devices this status does not make sense. Diagrams which represent the possible transitions from one status to another one have been drawn. For each transition, we have specified the different steps which need to be performed to make the change of status coherent and consistent. All these operations are done automatically.

The association between a device and a subsystem is an N:M relationship (a device can be part of N subsystems and a subsystem can contain M devices). For performance reasons, we have modeled the standard N:M relationship with a new representation based on prime numbers. To model an N:M relationship in the standard way, a new table must be created containing the relations. The method presented in this thesis avoids creating a new table by attributing a prime number as a unique identifier to the entity with the least number of distinct

values and adding an attribute (integer) to the other one. This attribute corresponds to the product of prime numbers of the possible associations. The algorithm works because the decomposition of a number in terms of its primes is unique. This application can be applied to any other N:M relationship such as STUDENT/COURSES.

To allow for easier network management, we have developed **routingtable_pck**, a PL/SQL package which generates routing and destination tables using connectivity information. The concepts of host and intermediate nodes have been defined. The main idea of the algorithm is to search for paths with some properties depending on whether it is used for a destination or a routing table. A path is a sequence of links. A routing path is a special path which starts with an intermediate node (switch) and ends at a host node (PCs). A destination path is a special path which ends at a host node (PCs). The algorithm has been successfully tested for different types of topologies. It does not generate looped paths (cycles). The **routingtable_pck** can be used in other fields which require the configuration of networks such as private companies or electrical facilities. This algorithm is used to generate the DHCP config file automatically, by creating the destination table of the given DHCP server. The formatting into specific terms used in the DHCP protocol is done with XML and XSLT. The method to create the DHCP config file is also generic enough to be applied in any other industrial context as it can be adapted for any given connectivity by the use of enabling/disabling a hardware equipment which needs an IP address.

Based on the same algorithm, we have extended it to get detailed paths between devices. It is used to compute the values of configurable device parameters or to check that the devices are properly connected.

Other PL/SQL codes have been implemented to avoid embedding long SQL statements.
The CIC DB and PL/SQL applications have been implemented to allow flexibility and consistency. They constitute an essential aspect in implementing an autonomics architecture. A change in the content of the CIC DB triggers updates which are essential to keep the information in the CIC DB consistent.
The other aspect was to build the object layer also according to the autonomics principles. CIC_DB_lib is a C-library which provides a set of functions to interact with the CIC DB. It has been built using an API which lists the different queries. With CIC_DB_lib, the user does not have to type any SQL queries or to have any knowledge of the CIC DB structure. The CIC_DB_lib has two bindings Python and PVSS built using respectively BOOST and GEH. The CIC_DB_lib provides cache mechanism which allows bulk collect insertion for performance reasons and automatic updates when possible. CIC_DB_lib and its two extensions have been tested to verify that the functions were operational and free of bugs (it includes memory leaks). Tests have been done to check that the functions were behaving correctly in case of a human error such as mistyping or inserting inconsistent information. The functions also try to minimize the human intervention by anticipating the reaction of the system in case this type of input is given. It was very useful for the inventory/history information which requires updating the data in a coherent manner.

Besides CIC_DB_lib, two Perl scripts have been built to automate the creation of the DHCP and DNS config files. It avoids mapping thousands of IP addresses with their corresponding host names and to make the association IP, MAC address and filename for a host name manually.

A PVSS library for recipes has been implemented by the CERN PVSS Support group. Here also, autonomics features have been applied by providing recipe template for electronics modules which are either commercial or widely used at the LHC. It avoids having errors in defining the configuration.

The object layer has also been built to permit to develop autonomics by reducing the human intervention.

On top of the object layer, there is the GUI layer which displays the information to the users. PVSS panels have been implemented using the PVSS extension of CIC_DB_lib and the PVSS library for recipes. It allows configuring electronics modules, retrieving a certain recipe and applying it to the equipment. There is also CDBVis, a Python tool which allows users to navigate through the CIC DB. It is based on the Python extension of CIC_DB_lib. It also permits to insert the connectivity of a system. CDBVis is a very useful tool as users can check that the connectivity has been properly inserted. In conjunction with the ECS, it can be used for fault detection. Here also the user interfaces have been designed according to the autonomics principles.
So far the different PVSS panels and the CDBVis have been built to be semi-autonomic. The running mode and the partition have still to be selected by the operator shift. The different recipes of devices have to be defined and the topology of a system has to be inserted by the user.
This project is used and will continue to be used in LHCb. In the future, table partitioning may be required to keep a good performance depending on the amount of data inserted.
In the future, the table schema can be extended with new parameters as the detector can evolve with years. More precise performance studies on the C-library implementation can be done. One possible extension is the history table. One can implement a table which contains the different mistakes which usually happen and associating them with a priority. Then a set of actions can be taken by the FSM according to the given priority. It also contributes in improving the autonomics architecture of the ECS, especially by improving its self-adaptability.

New features should be added to the CDBVis tool such as viewing the microscopic tool and also to integrate within PVSS. In other words, it should be possible to call CDBVis from PVSS. A filter should be proposed to the user when he views all the paths through a device. Some performance studies can be done to improve.

The ECS can integrate some more autonomics tools in the future, once the detector fully operational. For instance, before starting an experiment, a PVSS panel based on DIM and on the CIC_DB_lib to verify that the connectivity stored in the CIC_DB is complete, i.e. there is no link missing. There is a need to implement some protocols which enable to get the information of a board such as it serial number, MAC and IP addresses and also to which devices it is connected to. It can be very useful to check that the connectivity is complete.

Another subject possibility is to study the use of parallelism. In the current implementation, the CIC DB will be installed in an Oracle RAC (with 3 nodes). The controls PCs (less than one hundred) will load the required information from the CIC DB.

One can think to have smaller DB servers (slaves) which contain all the information (configuration, history/inventory, connectivity) related to a subsystem. They are replicated using Oracle Replication from the CIC DB master. The controls PCs will load the information from their respective DB slave server. Performance will be surely improved as queries will be performed against smaller tables and concurrent queries will be reduced.

Another possible extension of this project is to apply it in the context of tsunamis, earthquakes and the Grid. One can imagine a distributed database which contains all the characteristics of tsunamis, earthquakes, and any other natural catastrophes by regions of the world. It is similar to configuration information. It can also contain history information of a region, where the different natural damages occurred. It should also contain information on which towns or other regions can be affected if a given region is hit by a given catastrophe. This connectivity

information could be used to automatically send alerts to regions which can be affected if this type of catastrophe happens in this part of the world.

Appendices

Appendix A Proof of the equivalence of a routing path

A path P of length J is a routing path of length J

$$\Leftrightarrow \begin{pmatrix} W(P) = \sum_{i=1}^{i=J} W(L)_i = 2 \quad and \\ W(P) = \sum_{i=1}^{i=J-1} W(L)_i = 0 \end{pmatrix}$$

A path P of length J is a routing path of length J ⇔ P is a sequence of J+1 nodes which are all intermediate nodes (J nodes) except the terminal one which is a host node.
So P is a routing path of length J ⇔ P is a sequence of J-1 links between intermediate nodes and one link intermediate host nodes.
As the link weight between two intermediate nodes is equal to 0 and the link weight between an intermediate node and a host node is equal to 1, we have:
P is a routing path of length J⇔

$$\forall i \in [1, J-1], W(L)_i = 0 \quad and \quad W(L)_J = 1$$

As all the link weights are greater or equal than 0, we have:

$$\forall i \in [1, J-1], W(L)_i = 0 \Longleftrightarrow \sum_{i=1}^{i=J-1} W(L)_i = 0$$

and we also have

$$\forall i \in [1, J-1], W(L)_i = 0 \quad and \quad W(L)_J = 1$$
$$\Longleftrightarrow \sum_{i=1}^{i=J} W(L)_i = 1$$

Hence *a path P of length J is a routing path of length J*

$$\Leftrightarrow \begin{pmatrix} W(P) = \sum_{i=1}^{i=J} W(L)_i = 1 \quad and \\ W(P) = \sum_{i=1}^{i=J-1} W(L)_i = 0 \end{pmatrix}$$

Appendix B The interface of the PL/SQL package, routingtable_pck

```
/*************************************************************************
******/
// Author: L.Abadie
// version: v5.2
// PL/SQL package to generate and update routing and destination tables //
// automatically
/*************************************************************************
******/

create or replace package routingtable_pck AUTHID CURRENT_USER is

-- List of functions accessible to the users ---------------------
-- Function to call to generate the routingtable with the deviceid
      function createRT_all(devname number,path_necessary number default
0,round_trip number default 10,recreate_table number default 1) return
number;

-- Function to call to generate the routingtable with the devicename
function createRT_all_devname(devname varchar2,path_necessary number
default 0,round_trip number default 10,recreate_tab number default 1)
return number;

-- Function to call to generate the destinationtable with the devicename
      function createTD_all_devname(devname varchar2,path_necessary number
default 0,round_trip number default 10,recreate_tab number default 1)
return number;

-- Function to call to generate the destinationtable with the devicename
      function createTD_all(devname number,path_necessary number default
0,round_trip number default 10,recreate_tab number default 1) return
number;

-- List of internal functions not accessible by users --------------
-- create the AGGREGATED_LINKS and LINKPAIRS tables
      function      createNodeLink_tab(sysIDlist      number,recreate_table
number,bidi_needed number default 0) return number;

-- find all the valid routing/destination paths starting from the given ---
-- device (should be an intermediate node)
      function createPathTab(devfrom number,rt_needed number, round_trip
number default 10) return number;

-- find all the valid destination paths starting from the given device -- -
-- (should a host node)
      function   CreatePathTab_Host(   devfrom   number,rt_needed   number,
round_trip number default 10) return number;

-- Back to the port level and select the shortest path per destination and
-- store paths in a temporary table, ROUTINGPATHUSED=1 if shortest path
      function createRoutingTable_SP(devfrom number) return number ;

-- Back to the port level and store paths in a temporary table
      function createTabDestin(devfrom number) return number;

-- Insert the routing or destination paths from the temporary table to the
ROUTINGTABLE or DESTINATION table
```

```
      function           Insert_firsttab(devname           number,rt_needed
number,path_necessary number) return number;

-- Update the validity path further an update of a device, port or link --
-- status
      function UpdatePathUsed(systemname number) return number;

-- Create the AGGREGATED LINKs for the microscopic view
      procedure  createMicroNodeLink_tab(motherboardidList  in  out  number
,recreate_table number,rescode in out number, cpntid in out number,cpntname
varchar2 default 'none');
-- Find all the microscopic paths for a component
      function createMicroPathTab(devfrom number, round_trip number default
10) return number;

-- Find all the microscopic paths for an interface of the motherboard
      function  CreateMicroPathTab_Host(  devfrom_bis  number,  round_trip
number default 10) return number;

-- Select and Insert the microscopic paths starting from a given interface
-- to a component type
      function  InsertIntoMicroPathCnptType(  portid_from  number,cpnttype
varchar2,cpnttype_given number) return number;

-- Select and Insert the microscopic paths starting from a given interface
-- to another interface
      function  InsertIntoMicroPathFromPortID(  portid_from  number,portid
number, cpntname varchar2) return number;

-- Select and Insert the microscopic paths starting from a component to
-- an interface
      function  InsertIntoMicroPathUsingID(  cpntid  number,portid  number)
return number;

-- Delete the paths if links part of a path have been deleted
      function  DeleteLinksInPaths(pfromid  number,ptoid  number)  return
number;

-- check that the tables related to paths are update to date
      function checkUpdatePathTable(daq_sysid in number, tfc_sysid number)
return number ;

-- if not regenerate all the routing and destination tables
      function generKeyDTab(systemname in number) return number;

-- check that the tables related to paths are update to date and complete,
-- i.e. no routing or destination tables missing
      function CheckPathTabCompleteAndUpdate(systemname in number)  return
varchar2 ;

end routingtable_pck;
/
```

Appendix C Perl script to generate the dhcp config file

```
/*************************************************************************
******/
// Author: L.Abadie
// version: v1.2
// Perl script to generate the dhcp config file
// Replace XXX with the correct values
/*************************************************************************
******/
#!/usr/local/bin/perl

use DBI;
use XML::XSLT;
$user='XXXXX';
$passwd='XXXXXXX';
$dbh = DBI-> connect("dbi:Oracle:XXXXX","$user", "$passwd",{AutoCommit =>
0}) or die "Couldn't connect to database: " . DBI-> errstr," \n";


$dbh->{LongReadLen} = 512 * 1024;
$xmlfile_options="dhcp_options.xml";
$xmlfile = "dhcp_file.xml";
$xslfile = "style-test1.xsl";

$daq_sysID=5;
$dhcpconf="dhcpd.conf";
$dhcp_name=$ARGV[0];
print "dhcp_name= $dhcp_name";
if($dhcp_name eq "none")
{
      getAllHostNodes($xmlfile,$xmlfile_options);
      $dbh->disconnect;
      my $parser = XML::XSLT->new ($xslfile);
      $parser->transform($xmlfile);
      open(FILEHANDLE,">$dhcpconf")|| die("can't open datafile: $!");

      @output_file= $parser->toString;
      my $array_element ;
      foreach $array_element(@output_file)
      {
            print FILEHANDLE  $array_element." \n";
      }
      close(FILEHANDLE);
}
else
{
      $rescode_pl=0;
      $destin_temp=0;
      @list_tabname=CheckHostDestinationExistence($dhcp_name);
      $dhcp_devid=$list_tabname[1];

      if($list_tabname[0]==1)
      {
            $destin_temp=1;
            $rescode_pl=CreateLogicalView($daq_sysID);
            $rescode_pl=CreateDestinationTable($dhcp_devid);
      }
```

```
	if($rescode_pl==0)
	{

		#$SQLQuery =FormatQuery($destin_temp);

	getHostNodes($xmlfile,$dhcp_devid,$destin_temp,$xmlfile_options);
		$dbh->disconnect;
		my $parser = XML::XSLT->new ($xslfile);
		$parser->transform($xmlfile);
		open(FILEHANDLE,">$dhcpconf")|| die("can't open datafile: $!");

		@output_file= $parser->toString;
		my $array_element ;
		foreach $array_element(@output_file)
		{
			print FILEHANDLE  $array_element." \n";
		}
		close(FILEHANDLE);
		#open(FILEHANDLE,">$xmlfile")|| die("can't open datafile: $!");

		#close(FILEHANDLE);
	}
}
####################################subroutine   to   create   the   logical
view###################3
sub CreateLogicalView{
    my $subsystemID=$_[0];
    my $rescode=-1;
    my                                                    $plsql_query="begin
:rescode:=routingtable_pck.createnodelink_tab(:sysID,1,0); end;";
    my $csr=$dbh->prepare($plsql_query);
    $csr->bind_param_inout(":sysID",\$subsystemID,100);
    $csr->bind_param_inout(":rescode", \$rescode, 100);
    $csr->execute();
    return $rescode;

}
####################################Subroutines bodies#####################
#sub routine which  creates the destination table of the dhcp server name
given... don't forget
sub CreateDestinationTable{
my $rescode=1;
my $dhcpname=$_[0];

$pl_sql_query="begin   :rescode:=routingtable_pck.CreatePathTab_Host(:devid,
0); if :rescode=0 then :rescode:=routingtable_pck.createTabDestin(:devid);
end if; end;";
my $csr = $dbh->prepare($pl_sql_query);

	# The value of $devid is _copied_ here
	$csr->bind_param_inout(":devid",\$dhcpname,100);

	$csr->bind_param_inout(":rescode", \$rescode, 100);

	# The execute will automagically update the value of $is_odd
	$csr->execute();


return $rescode;
```

```
}
####################################Subroutines bodies#####################
#sub routine which returns first if we need to create the destin table to
check dchp server connectivity, then the #deviceid
sub CheckHostDestinationExistence{
my $deviceid;
my $result_temp=0;
my @list_subsystem;
my $createtab_needed=0;

my             $SQLquery='select             count(pathid)             from
lhcb_destinationtable,lhcb_lg_devices   where   nodeid_start0=deviceid   and
devicename=:1 and rownum=1';
my $sth=$dbh->prepare($SQLquery);
$sth->bind_param(1,$_[0]);
$sth->execute;
while(@deviceid_list=$sth->fetchrow_array())
    {
      $result_temp=$deviceid_list[0];
            if($result_temp==0)
      {
                  $createtab_needed=1;
      }
    }
push(@list_subsystem,$createtab_needed);
my $SQLquery='select deviceid from lhcb_lg_devices where devicename=:1 ';
my $sth=$dbh->prepare($SQLquery);
$sth->bind_param(1,$_[0]);
$sth->execute;
while(@deviceid_list=$sth->fetchrow_array())
    {
      $deviceid=$deviceid_list[0];
    }
push(@list_subsystem,$deviceid);

return @list_subsystem;
}
#######subroutine which formats the SQL query using XML attributes#########

sub FormatQuery
{
if ($_[0]==0)
{
my                  $SQLstatement                    =                  'select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,\'.\',1
)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from      lhcb_ipinfo      e,lhcb_port_properties
t,lhcb_device_booting  r,  lhcb_hwport_properties  s,lhcb_lg_devices  g  where
t.ipaddress=e.ipaddress and r.deviceid=g.deviceid and t.portid in (select
ptoid1 from lhcb_destinationtable where pathused=1 and NODEID_START0=:1)
and     t.port_nbr=s.port_nbr     and     t.port_type=s.port_type     and
t.deviceid=g.deviceid     and     s.serialnb=g.serialnb     union     select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,\'.\',1
)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from      lhcb_ipinfo      e,lhcb_port_properties
t,lhcb_devicetype_booting   r,lhcb_hwport_properties   s,lhcb_lg_devices   g
where   t.ipaddress=e.ipaddress   and   r.devicetypeid=g.devicetypeid   and
```

```
t.portid in (select ptoid1 from lhcb_destinationtable where pathused=1 and
NODEID_START0=:nodeid)         and         t.port_nbr=s.port_nbr         and
t.port_type=s.port_type       and             t.deviceid=g.deviceid       and
s.serialnb=g.serialnb   and   g.deviceid   not   in   (select   deviceid   from
lhcb_device_booting)';
}
else
{
my                  $SQLstatement                    =                   'select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(ipname,1,instr(e.ipname,\'.\',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from       lhcb_ipinfo       e,lhcb_port_properties
t,lhcb_device_booting  r,  lhcb_hwport_properties  s,lhcb_lg_devices  g  where
t.ipaddress=e.ipaddress and r.deviceid=g.deviceid and t.portid in (select
ptoid1 from lhcb_destintab_temp where pathused=1 and NODEID_START0=:1) and
t.port_nbr=s.port_nbr and t.port_type=s.port_type and s.deviceid=t.deviceid
and             t.deviceid=g.deviceid             union             select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(ipname,1,instr(e.ipname,\'.\',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from       lhcb_ipinfo       e,lhcb_port_properties
t,lhcb_devicetype_booting  r,  lhcb_hwport_properties  s,lhcb_lg_devices  g
where   t.ipaddress=e.ipaddress   and   r.devicetypeid=g.devicetypeid   and
t.portid in (select ptoid1 from lhcb_destintab_temp where pathused=1 and
NODEID_START0=:nodeid)         and         t.port_nbr=s.port_nbr         and
t.port_type=s.port_type and s.deviceid=t.deviceid and t.deviceid=g.deviceid
and g.deviceid not in (select deviceid from lhcb_device_booting)';
}
return $SQLstatement;
}

############  subroutine  which  prints  the  results  in  the  xml  file
##############
sub getHostNodes {
    my $j=0;
  open(XMLOPTIONS,"$_[3]") || die("can't open datafile: $!");
  @lines=<XMLOPTIONS>;
  close(XMLOPTIONS);
  open(FILEHANDLE,">$_[0]") || die("can't open datafile: $!");
  print FILEHANDLE "<?xml version = '1.0'?> \n";
  print FILEHANDLE "<page> \n";
  foreach $option_dhcp(@lines)
  {
      print                                                          FILEHANDLE
"<options><option>$option_dhcp</option></options>\n";
      print "options= $option_dhcp";
  }



my $sth;

if ($_[2]==0)
{
$sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,'.',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from       lhcb_ipinfo       e,lhcb_port_properties
t,lhcb_device_booting  r,  lhcb_hwport_properties  s,lhcb_lg_devices  g  where
```

```
t.ipaddress=e.ipaddress and r.deviceid=g.deviceid and t.portid in (select
ptoid1 from lhcb_destinationtable where pathused=1 and nodeid_start0=:1)
and t.port_nbr=s.port_nbr and t.port_type=s.port_type and
t.deviceid=g.deviceid and s.serialnb=g.serialnb union all select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,'.',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info from lhcb_ipinfo e,lhcb_port_properties
t,lhcb_devicetype_booting r,lhcb_hwport_properties s,lhcb_lg_devices g
where t.ipaddress=e.ipaddress and r.devicetypeid=g.devicetypeid and
t.portid in (select ptoid1 from lhcb_destinationtable where pathused=1 and
NODEID_START0=:2) and t.port_nbr=s.port_nbr and t.port_type=s.port_type and
t.deviceid=g.deviceid and s.serialnb=g.serialnb and g.deviceid not in
(select deviceid from lhcb_device_booting) });
}
else
{
$sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(ipname,1,instr(e.ipname,'.',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info from lhcb_ipinfo e,lhcb_port_properties
t,lhcb_device_booting r, lhcb_hwport_properties s,lhcb_lg_devices g where
t.ipaddress=e.ipaddress and r.deviceid=g.deviceid and t.portid in (select
ptoid1 from lhcb_destintab_temp where pathused=1 and nodeid_start0=:1) and
t.port_nbr=s.port_nbr and t.port_type=s.port_type and s.serialnb=g.serialnb
and t.deviceid=g.deviceid union all select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,'.',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info from lhcb_ipinfo e,lhcb_port_properties
t,lhcb_devicetype_booting r,lhcb_hwport_properties s,lhcb_lg_devices g
where t.ipaddress=e.ipaddress and r.devicetypeid=g.devicetypeid and
t.portid in (select ptoid1 from lhcb_destintab_temp where pathused=1 and
NODEID_START0=:2) and t.port_nbr=s.port_nbr and t.port_type=s.port_type and
t.deviceid=g.deviceid and s.serialnb=g.serialnb and g.deviceid not in
(select deviceid from lhcb_device_booting) });
}
$sth->bind_param(1,$_[1]);
$sth->bind_param(2,$_[1]);
$sth->execute();
```

```
  my $r=0;
  my $i=0;
  my $count_distinct_subnet=0;
  my $count_distinct_subnet1=0;
```

```
  my @distinct_subnetID;
  my @distinct_subnetmask;
   my @res_query;
  while(@data=$sth->fetchrow_array())
    {

      my $line_get=$data[0];
      my $subnet_mask=$data[2];
      $SubnetID=GetSubnetID($data[1],$data[2]);
```

```
      $data[1]=$subnetID;
      $_= $SubnetID;
      $count_distinct_subnet1= $count_distinct_subnet;
      push(@distinct_subnetID, $_) unless ($seen{$_}++);
      $count_distinct_subnet=@distinct_subnetID;
      if($count_distinct_subnet>$count_distinct_subnet1)
      {
        push(@distinct_subnetmask,$data[2]);
      }
      push(@res_query,$line_get."||".$SubnetID);
    }

  foreach(@distinct_subnetID)
    {
      $SubnetID=$distinct_subnetID[$r];
      print FILEHANDLE "<subnet> \n";
      print FILEHANDLE "<subnetID> ".$SubnetID." </subnetID> \n";
      print    FILEHANDLE    "<subnet_mask>    ".$distinct_subnetmask[$r]."
</subnet_mask> \n";
      print FILEHANDLE "<rowset> \n";
      $i=0;

      foreach(@res_query)
      {
       @data=split(/\|\|/,$res_query[$i]);

       if($data[1] eq $SubnetID)
          {
             $line_get=$data[0];
             print FILEHANDLE $line_get." \n";
          }
        $i++;
      }
       print FILEHANDLE "</rowset> \n";
      $r++;
       print FILEHANDLE "</subnet> \n";
    }



  print FILEHANDLE "</page> \n";
  close(FILEHANDLE);
$dbh->commit();

}
############subroutine which returns the subnet ID ####################
sub GetSubnetID
{
  my @ip_add = split(/\./,$_[0]);
  my @subnet_ip=split(/\./,$_[1]);
  my $i=0;
  my $temp1='';
  my $bin1='';
  my $temp2='';
  my $bin2='';
  my $binary_ip='';
  my $binary_subnet='';
  my $res1='';
  my $res2='';

  foreach (@ip_add)
```

```
  {
   $temp1=sprintf "%d",$ip_add[$i];

   $bin1=unpack("B32", pack("N", $temp1));

   $temp2=sprintf "%d",$subnet_ip[$i];
   $bin2=unpack("B32", pack("N", $temp2));
   $res1=ProceedSubnetID($bin1,$bin2);
   if($i>0)
     {
       $res2=$res2."\.".$res1;
     }
   else
     {
        $res2=$res1;

     }
   $i++;
  }


  return $res2;
}
#####subroutine which multiply the 2 vectors as .* (Matlab)###############
sub ProceedSubnetID
{

  my $l=0;
  my $k=0;
  my $res1=0;
  my $i=0;
  my @bin_add=split(//,$_[0]);
  my @bin_subnet=split(//,$_[1]);
  my $j=@bin_add-1;
  foreach(@bin_add)
    {
      $l=$bin_add[$i]*$bin_subnet[$i];
      $k=sprintf "%d",$l;
      if($k==1)
      {
        $res1=$res1+2**$j;
      }
      $i++;
      $j--;
    }
  return $res1;
}

############ subroutine which prints the results in the xml file
##############
sub getAllHostNodes {
    my $j=0;
  open(XMLOPTIONS,"$_[1]") || die("can't open datafile: $!");
  @lines=<XMLOPTIONS>;
  close(XMLOPTIONS);
  open(FILEHANDLE,">$_[0]") || die("can't open datafile: $!");
  print FILEHANDLE "<?xml version = '1.0'?> \n";
  print FILEHANDLE "<page> \n";
  foreach $option_dhcp(@lines)
  {
```

```
      print                                                    FILEHANDLE
"<options><option>$option_dhcp</option></options>\n";
      print "all host nodes options= $option_dhcp";
  }



my $sth;


$sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,'.',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from      lhcb_ipinfo      e,lhcb_port_properties
t,lhcb_device_booting  r,  lhcb_hwport_properties  s,lhcb_lg_devices  g  where
t.ipaddress=e.ipaddress and r.deviceid=g.deviceid and t.port_nbr=s.port_nbr
and    t.port_type=s.port_type    and         t.deviceid=g.deviceid    and
s.serialnb=g.serialnb  and  g.node=1  and  t.port_type='control'  union  all
select
xmlelement("row",xmlelement("ethernet_add",s.macaddress),xmlelement("ip_add
",t.ipaddress),xmlelement("ipname",substr(e.ipname,1,instr(e.ipname,'.',1)-
1)),xmlelement("filename",r.boot_image_location)).getClobVal(),t.ipaddress,
e.subnet_info             from      lhcb_ipinfo      e,lhcb_port_properties
t,lhcb_devicetype_booting   r,lhcb_hwport_properties   s,lhcb_lg_devices   g
where   t.ipaddress=e.ipaddress   and   r.devicetypeid=g.devicetypeid   and
g.node=1   and   t.port_type='control'   and   t.port_nbr=s.port_nbr   and
t.port_type=s.port_type      and            t.deviceid=g.deviceid      and
s.serialnb=g.serialnb  and  g.deviceid  not  in  (select  deviceid  from
lhcb_device_booting) });


$sth->execute();



  my $r=0;
  my $i=0;
  my $count_distinct_subnet=0;
  my $count_distinct_subnet1=0;




  my @distinct_subnetID;
  my @distinct_subnetmask;
   my @res_query;
  while(@data=$sth->fetchrow_array())
    {

      my $line_get=$data[0];
      my $subnet_mask=$data[2];
      $SubnetID=GetSubnetID($data[1],$data[2]);
      $data[1]=$subnetID;
      $_= $SubnetID;
      $count_distinct_subnet1= $count_distinct_subnet;
      push(@distinct_subnetID, $_) unless ($seen{$_}++);
      $count_distinct_subnet=@distinct_subnetID;
      if($count_distinct_subnet>$count_distinct_subnet1)
      {
        push(@distinct_subnetmask,$data[2]);
```

```
      #print FILEHANDLE "<subnet_found> $SubnetID <subnet_found>\n";
      }
      push(@res_query,$line_get."||".$SubnetID);

    }

  foreach(@distinct_subnetID)
    {
      $SubnetID=$distinct_subnetID[$r];
      print FILEHANDLE "<subnet> \n";
      print FILEHANDLE "<subnetID> ".$SubnetID." </subnetID> \n";
      print    FILEHANDLE    "<subnet_mask>    ".$distinct_subnetmask[$r]."
</subnet_mask> \n";
      print FILEHANDLE "<rowset> \n";
      $i=0;
      foreach(@res_query)
      {
       @data=split(/\|\|/,$res_query[$i]);

       if($data[1] eq $SubnetID)
          {
             $line_get=$data[0];
             print FILEHANDLE $line_get." \n";
          }
        $i++;
      }
       print FILEHANDLE "</rowset> \n";
      $r++;
       print FILEHANDLE "</subnet> \n";
    }



  print FILEHANDLE "</page> \n";
  close(FILEHANDLE);
$dbh->commit();

}
```

Appendix D Example of C code to exclude and include back host nodes

```
/*************************************************************************
******/
// Author: L.Abadie
// compatible with version: v3.2 of the CIC_DB_lib
// example of C code to exclude and include back nodes
/*************************************************************************
******/

#include <iostream>
#include<stdio.h>
#include<unistd.h>
#include <ctime>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "cicDB.h"

int execute_dhcp(char* dhcp_name)
{
  int res=0;
  execl("/usr/local/bin/perl","perl","dhcpCfg_generate.pl",dhcp_name,NULL);
  std::cout<<"could not execute the perl script and err="<<std::endl;
  res=-1;
  return res;
}

int main()
{
  pid_t pid_pere,pid_fils,pid_first,pid_1;
  int res=0;
  int res2=0;
  int len_array=1000;
  char* nto_list;
  char ErrMess[1000];
  int status=0;
  int status1=0;
  int i=0;
 char nodeName[50];
char nodeName_bis[50];

char dhcp_server1[50]="DAQ_CTRL_PC_01" ;
char dhcp_server2[50]="DAQ_CTRL_PC_02" ;

  res=DBConnexion(DB_NAME,login,pwd,ErrMess);
  for(i=1;i<33;i++)
 {
//disabling the host nodes from node_2_1 to node_2_32
sprintf(nodeName,"node_2_%d",i);
      if(i = =1)
      // the second parameter (nodeused) is set to 0 to disable the node.
      res_query=UpdateMultipleDeviceNodeUsed(nodeName,0,1,0,ErrMess);
else
      {
            if(i==32)
```

```
            resquery=UpdateMultipleDeviceNodeUsed(nodeName,0,0,1,ErrMess);
            else
            resquery=UpdateMultipleDeviceNodeUsed(nodeName,0,0,0,ErrMess);
      }
}
pid_first=fork();
if(pid_first==-1)
    {
      perror("error fork");
      res=DBDeconnexion(ErrMess);
      exit(1);
    }
  else
    {
       pid_fils=getpid();
       pid_pere=getpid();
       if(pid_first==0)
       {
         std::cout<<"before generating the first dhpc config file
..."<<std::endl;
         res=execute_dhcp(dhcp_server_1);
         std::cout<<"Error in generating the first dhpc config
file..."<<std::endl;
         res=DBDeconnexion(ErrMess);
         perror("exec");
         exit(1);
       }
       else
       {
         res=wait(&status);
         if(WIFEXITED(status)!=0)
         {
            for(i=1;i<33;i++)
            {
                        sprintf(nodeName,"node_2_%d",i);
                        sprintf(nodeName_bis,"node_1_%d",i);
                        if(i = =1)
                        {

      res_query=UpdateMultipleDeviceNodeUsed(nodeName,1,1,0,ErrMess);

      res_query=UpdateMultipleDeviceNodeUsed(nodeName_bis,0,0,0,ErrMess);
                        }
                        else
                        {
                              if(i==32)
                              {

      resquery=UpdateMultipleDeviceNodeUsed(nodeName,1,0,0,ErrMess);

      resquery=UpdateMultipleDeviceNodeUsed(nodeName_bis,0,0,1,ErrMess);
                              }
                              else
                              {

      resquery=UpdateMultipleDeviceNodeUsed(nodeName,1,0,0,ErrMess);

      resquery=UpdateMultipleDeviceNodeUsed(nodeName_bis,0,0,0,ErrMess);
                              }
                        }
            }
```

```
            pid_1=fork();
            if(pid_1= =-1)
              {
                perror("error fork");
                res=DBDeconnexion(ErrMess);
                exit(1);
              }
            else
              {
                if(pid_1==0)
                  {
                  std::cout<<"before generating the second dhcp cfg file
..."<<std::endl;
                  res=execute_dhcp(dhcp_server_2);
                  std::cout<<"Error in generating the second dhpc config
file..."<<std::endl;

                  res=DBDeconnexion(ErrMess);
                  perror("exec");
                  exit(1);
                  }
                else
                  {
                       res=wait(&status1);
                       if(WIFEXITED(status1)!=0)
                        {
                              for(i=1;i<33;i++)
                             {
      sprintf(nodeName,"node_1_%d",i);
      if(i = =1)
      // the second parameter (nodeused) is set to 0 to disable the node.
      res_query=UpdateMultipleDeviceNodeUsed(nodeName,1,1,0,ErrMess);
else
      {
            if(i==32)
            resquery=UpdateMultipleDeviceNodeUsed(nodeName,1,0,1,ErrMess);
            else
            resquery=UpdateMultipleDeviceNodeUsed(nodeName,1,0,0,ErrMess);
      }
}


                        res=DBDeconnexion(ErrMess);
                         }
                         else
                         {
                               std::cout<<"child     process     2     exited
abnormally..."<<std::endl;
                               res=DBDeconnexion(ErrMess);
                         }
                  }

              }


             }
          else
          {
             std::cout<<"child     process     1     did     not     exit
properly..."<<std::endl;
             res=DBDeconnexion(ErrMess);
```

```
                perror("exec");
                exit(1);
            }
        }
    }
}
```

Appendix E Perl script to generate the dns set of files

```
/************************************************************************
******/
// Author: L.Abadie
// version: v1.2
// Perl script to generate the dns forward and reverse file
// Replace XXX with the correct values
/************************************************************************
******/

#!/usr/local/bin/perl

use DBI;
use XML::XSLT;
use Fcntl;
$user='XXXXX';
$passwd='XXXXXX';
$lhcb_zone_name="XXXX";
$dns_name_master="XXXXXXXXX";

$port_type='control';
$dns_fct='DNS_SERVER';
$max_limit_creation=9;
$xmlfile_options="dns_options.xml";
$xmlfile_dns = "ecs.lhcb.xml";
$dns_configfile = "forward.ecs.lhcb";
$local_host="root.localhost.";
$xslfile_dns = "dns_style.xsl";
$xslfile_reverse = "dns_style_rev.xsl";
$serial_dns=1;
my @list_subnetID;
my $serial_dns=GetNextSerial($xmlfile_dns,$max_limit_creation);
if( $serial_dns==-1)
{
      print "ERROR : dns config have been created more than
$max_limit_creation \n";
      exit;
}
else
{
      $dbh        =        DBI->        connect("dbi:Oracle:devdb10","$user",
"$passwd",{AutoCommit => 0}) or die "Couldn't connect to database: " . DBI-
> errstr," \n";
      $dbh->{LongReadLen} = 512 * 1024;
      #generate the forward dns config file
      $rescode=getIPnameIPaddress($xmlfile_dns,$xmlfile_options,$dns_name_m
aster,$lhcb_zone_name,$serial_dns,$port_type,$dns_fct,$local_host);
      my $parser = XML::XSLT->new ($xslfile_dns);
      $parser->transform($xmlfile_dns);
      if (-e $reverse_dns_file)
      {
            open(FILEHANDLE,">$dns_configfile")     ||     die("can't     open
datafile: $!");
      }
      else
      {
            sysopen(FILEHANDLE,"$dns_configfile",  O_WRONLY|O_TRUNC|O_CREAT)
|| die("can't open datafile 2: $!");
```

```
	}

	@output_file= $parser->toString;
	my $array_element ;
	foreach $array_element(@output_file)
	{
		print FILEHANDLE  $array_element." \n";
	}
	close(FILEHANDLE);

	if($rescode==0)
	{
		my $i=0;

		#generate the reverse dns config file per subnet;


	@iplist_content=getIPaddressIPname($subID_dns_file,$xmlfile_options,$
dns_name_master,$serial_dns,$lhcb_zone_name,$port_type,$dns_fct,$subID,$loc
al_host,$reverse_dns_file);
		$size2 = $iplist_content[0];
		if($size2!=-1)
		{
			$iplist_len = @iplist_content;
			$subID_line=$iplist_content[$iplist_len-1];
			@list_subnetID=split(/:/,$subID_line);
			pop(@iplist_content);

			foreach $subID(@list_subnetID)
			{
				my @ipdecomp = split(/\./,$subID);
				if($ipdecomp[1]==0)
				{
					$subID_dns_file=$ipdecomp[0].".in-
addr.arpa.xml";
					$reverse_dns_file=$ipdecomp[0].".in-
addr.arpa";
				}
				else
				{
					if($ipdecomp[2]==0)
					{

	$subID_dns=join(".",$ipdecomp[0],$ipdecomp[1]);
						$subID_dns_file=$subID_dns.".in-
addr.arpa.xml";
						$reverse_dns_file=$subID_dns.".in-
addr.arpa";
					}
					else
					{
						if($ipdecomp[3]==0)
						{

	$subID_dns=join(".",$ipdecomp[0],$ipdecomp[1],$ipdecomp[2]);
							$subID_dns_file=$subID_dns.".in-
addr.arpa.xml";
							$reverse_dns_file=$subID_dns.".in-
addr.arpa";
						}
					}
```

```
                        }

      @generic_settings=getIPaddressIPnameForDNS($subID_dns_file,$xmlfile_o
ptions,$dns_name_master,$serial_dns,$lhcb_zone_name,$port_type,$dns_fct,$su
bID,$local_host,$reverse_dns_file);
                        $size1 = $generic_settings[0];
                        if(size1!=-1)
                        {

                              if (-e $subID_dns_file)
                              {
                                    #print  "file  exist  and  $subID_dns_file
\n";
                                    open(FILEHANDLE,">$subID_dns_file")    ||
die("can't open datafile: $!");
                              }
                              else
                              {
                                    #print   "file   doesn   't   exist   and
$subID_dns_file \n";
                                    sysopen(FILEHANDLE,"$subID_dns_file",
O_WRONLY|O_TRUNC|O_CREAT) || die("can't open datafile 2: $!");
                              }
                              foreach $generic_line(@generic_settings)
                              {
                                    print FILEHANDLE  $generic_line." \n";
                              }

                              foreach $ip_line(@iplist_content)
                              {
                                     @ipline_split = split(/\?/,$ip_line);
                                     $current_subID=$ipline_split[0];

                                    if($subID=~m /$current_subID/ )
                                    {
                                           $current_ipline=$ipline_split[1];
                                          print                    FILEHANDLE
$current_ipline." \n";
                                    }
                              }
                              print FILEHANDLE "</rowset> \n";
                              print FILEHANDLE "</page> \n";
                              close(FILEHANDLE);
                              my       $parser       =       XML::XSLT->new
($xslfile_reverse);
                              $parser->transform($subID_dns_file);
                              if (-e $reverse_dns_file)
                              {
                                    open(FILEHANDLE,">$reverse_dns_file") ||
die("can't open datafile: $!");
                              }
                              else
                              {
                                    sysopen(FILEHANDLE,"$reverse_dns_file",
O_WRONLY|O_TRUNC|O_CREAT) || die("can't open datafile 2: $!");
                              }
                              @output_file= $parser->toString;
                              my $array_element ;
                              foreach $array_element(@output_file)
                              {
                                    print FILEHANDLE  $array_element." \n";
```

```
                              }
                              close(FILEHANDLE);
                        }
                        else
                        {
                              print "ERROR in creating reverser dns_file for
the following file  $subID_dns_file, no IP address was found";
                        }
                  }
            }
            else
            {
                  print  "ERROR  in  creating  reverser  dns_file,  no  dns
found";
                  $rescode=$dbh->disconnect();
                  exit;
            }
            print " OUT OF LOOP \n";
      }
      else
      {
            print "ERROR in creating dns_file";
            $rescode=$dbh->disconnect();
            exit;
      }
      print "just before disconnect \n";
      $rescode=$dbh->disconnect();
      print " disconnect \n";
      exit;
}

#######subroutine which returns the next serial#########

sub GetNextSerial
{
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =localtime(time);
$mon+=1;
$year += 1900;
$stop=0;
my $last_serial=0;
#my $serial=$year.$mon.$mday."0";
my $myserial=sprintf("%d%02d%02d0",$year,$mon,$mday);
my $return_serial;
#print "my option 0 $myserial \n";
if (-e $_[0])
{
      open(XMLDNS,"$_[0]") || die("can't open datafile: $!");
      @lines=<XMLDNS>;
      foreach $option_dns(@lines)
      {
            if($option_dns =~/<serial>/i)
            {
                  #print "my option 1 $option_dns \n";
                  my        $option_line        =        substr($option_dns,
index($data,"<serial>")+9,index($data,"</serial>")-
index($data,"<serial>")+9);
                  print "my option 2 $option_line \n";
                  $stop=1;
                  $last_serial=sprintf("%d",$option_line);
                  #print "my option 3 $last_serial \n";
                  last;
```

```
            }

      }

      if($stop==0)
      {
            $last_serial=0;
      }
      #print "my option 4  $myserial and $last_serial \n";
      if($myserial<=$last_serial )
      {
            #print "my option 5 $last_serial and $_[1] \n";
            if(($last_serial-$_[1])%10!=0)
            {
                  $last_serial++;
                  #print "my option 6 $last_serial \n";
                  $return_serial=$last_serial;

            }
            else
            {
                  $return_serial=-1;
            }
      }
      else
      {
            $return_serial=$myserial;
      }
}
else
{
      $return_serial=$myserial;
}
return $return_serial;
}



#######

############  subroutine  which  prints  the  results  in  the  xml  file
##############
sub getIPnameIPaddress {
    my $j=0;
    my $rescode=0;
    my $sth_1;
      my $first_line="$_[3]. IN SOA $_[2]."."$_[3]. ".  "$_[7] ";
      my $choice=0;


  open(XMLOPTIONS,"$_[1]") || die("can't open datafile: $!");
  @lines=<XMLOPTIONS>;
  close(XMLOPTIONS);
  print "file name $_[0]";
  if (-e $subID_dns_file)
      {
            open(FILEHANDLE,">$_[0]") || die("can't open datafile: $!");
      }
      else
```

```
      {
            sysopen(FILEHANDLE,"$_[0]",     O_WRONLY|O_TRUNC|O_CREAT)     ||
die("can't open datafile 2: $!");
      }
  print FILEHANDLE "<?xml version = '1.0'?> \n";
  print FILEHANDLE "<page> \n";
  print FILEHANDLE "<option_list>\n";
  print FILEHANDLE "<SOA>${first_line}</SOA>\n";
   print FILEHANDLE "<serial>${_[4]}</serial>\n";
  foreach $option_dns(@lines)
  {
      print FILEHANDLE "$option_dns \n";
  }

  print FILEHANDLE "</option_list>\n";

#my $statement=FormatQuery($choice);
$sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ipname",e.ipname),xmlelement("ipadd",t.ipaddre
ss),xmlelement("function",'NS')).getClobVal()        from        lhcb_ipinfo
e,lhcb_port_properties  t,lhcb_lg_devices  g,lhcb_device_functions  l  where
t.ipaddress=e.ipaddress  and  t.deviceid=g.deviceid  and  t.port_type=:1  and
l.function_name=:2       and       mod(g.functionid,l.functionid)=0       and
g.functionid!=0});
$sth->bind_param(1,$_[5]);
$sth->bind_param(2,$_[6]);
$sth->execute();
  print FILEHANDLE "<rowset> \n";
print                                                                FILEHANDLE
"<row><ipname>$_[3].</ipname><ipadd>$_[2].$_[3].</ipadd><function>NS</funct
ion></row>\n";
   my $zone_suffix=".".$_[3];
  while(@data=$sth->fetchrow_array())
  {


            if($data[0]=~/${zone_suffix}<\/ipname>/i)
            {

                  $data[0]=~ s/$zone_suffix/ /;
                  print FILEHANDLE "$data[0]\n";
            }



      $j++;


  }

 $sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ipname",e.ipname),xmlelement("ipadd",t.ipaddre
ss),xmlelement("function",'A')).getClobVal()        from        lhcb_ipinfo
e,lhcb_port_properties  t,lhcb_lg_devices  g,lhcb_device_functions  l  where
t.ipaddress=e.ipaddress  and  t.deviceid=g.deviceid  and  t.port_type=:1  and
l.function_name=:2                                                    and
mod(decode(g.functionid,0,l.functionid+1,g.functionid),l.functionid)!=0});


$sth->bind_param(1,$_[5]);
$sth->bind_param(2,$_[6]);
```

```
$sth->execute();

   my $zone_suffix=".".$_[3];
  while(@data=$sth->fetchrow_array())
  {


            if($data[0]=~/${zone_suffix}<\/ipname>/i)
            {

                  $data[0]=~ s/$zone_suffix/ /;
                  print FILEHANDLE "$data[0]\n";
            }



      $j++;


  }
  if($j==0)
  {
      #print "ERROR : no ip address \n";
      $rescode=-1;
      close(FILEHANDLE);
      return $rescode;
  }
  $choice++;
  #print "second query \n";

  $sth=$dbh->prepare(q{select
xmlelement("row_scd",xmlelement("cipname",e.ipname),xmlelement("cname",g.ip
alias)).getClobVal()        from        lhcb_ipinfo        e,lhcb_ipaliases
g,lhcb_port_properties  m  where  g.ipaddress=e.ipaddress  and  m.port_type=:1
and m.ipaddress=e.ipaddress});

   $sth->bind_param(1,$_[5]);
   $sth->execute();
   $j=0;
  while(@data=$sth->fetchrow_array())
  {
       #print "line $data[0]  \n";
       if($data[0]=~ /${zone_suffix}<\/cipname>/i)
       {
            #print "in cname data =$data[0] \n";
            $data[0]=~ s/$zone_suffix/ /;
       }
       if($data[0]=~/${zone_suffix}<\/cname>/i)
       {
            #print "in cname data =$data[0] \n";
            $data[0]=~ s/$zone_suffix/ /;
       }
       print FILEHANDLE "$data[0]  \n";


  }
  print FILEHANDLE "</rowset> \n";
  print FILEHANDLE "</page> \n";
  close(FILEHANDLE);

return $rescode;
```

```
}


############ get the common header to the reverse dns file #############
sub getIPaddressIPnameForDNS {
    my $j=0;
    my $sth_1;
    my $rescode=0;
    my $serial_dns=$_[3];
     my $zone_suffix=".".$_[4];
    my $choice=2;
    my @header_content;
      my $first_line="$_[9]. IN SOA ".$_[2].$zone_suffix.".  ".$_[8] ;
  open(XMLOPTIONS,"$_[1]") || die("can't open datafile: $!");
  @lines=<XMLOPTIONS>;
  close(XMLOPTIONS);
  my $new_line="<SOA>${first_line}</SOA>";
  push(@header_content,"<?xml version = '1.0'?> ");
  push(@header_content, "<page>");
  push(@header_content,"<option_list>");
  push(@header_content, $new_line);
  $new_line="<serial>${serial_dns}</serial>";
  push(@header_content, $new_line);
  foreach $option_dns(@lines)
  {
      push(@header_content,${option_dns});
     #print "in the reverse options= $option_dns";
  }
  push(@header_content, "</option_list>");

 #my $statement=FormatQuery($choice);
 $sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ipadd",t.ipaddress),xmlelement("ipname",e.ipna
me||'.'),xmlelement("function",'NS')).getClobVal(),e.subnet_info        from
lhcb_ipinfo             e,lhcb_port_properties             t,lhcb_lg_devices
g,lhcb_device_functions     l     where     t.ipaddress=e.ipaddress     and
t.deviceid=g.deviceid   and   t.port_type=:1   and   l.function_name=:2   and
mod(g.functionid,l.functionid)=0 and g.functionid!=0});



$sth->bind_param(1,$_[5]);
$sth->bind_param(2,$_[6]);
$sth->execute();



  push(@header_content, "<rowset>");

$new_line="<row><ipadd>$_[9].</ipadd><ipname>$_[2]$zone_suffix.</ipname><fu
nction>NS</function></row>";
  push(@header_content, $new_line);
  while(@data=$sth->fetchrow_array())
  {



                  my           $ipadd           =           substr($data[0],
index($data[0],"<ipadd>")+7,index($data[0],"</ipadd>")-
index($data[0],"<ipadd>")-7);
```

```
                    my @ipdecomp = split(/\./,$ipadd);

                    @ipdecomp=reverse @ipdecomp;

                    my $reverse_ip=join(".",@ipdecomp);
                    $reverse_ip=$reverse_ip.".";

                    $data[0]=~ s/$ipadd/$reverse_ip/;
                    push(@header_content,$data[0] );



      $j++;

  }
  if($j==0)
  {
    print "ERROR no ipaddress \n";
      $rescode=-1;
      push(@header_content, $rescode);
      return @header_content;
  }
  else
  {
      return @header_content;
  }
 }
############ subroutine which prints the results in the xml file
##############
sub getIPaddressIPname {
    my $j=0;
    my $sth_1;
    my $rescode=0;
    my $serial_dns=$_[3];
    my $zone_suffix=".".$_[4];
    my $choice=2;
      my @iplist;
      my @distinct_subnetID;
  my $count_distinct_subnet=0;
  $sth=$dbh->prepare(q{select
xmlelement("row",xmlelement("ipadd",t.ipaddress),xmlelement("ipname",e.ipna
me||'.'),xmlelement("function",'PTR')).getClobVal(),e.subnet_info      from
lhcb_ipinfo             e,lhcb_port_properties             t,lhcb_lg_devices
g,lhcb_device_functions     l     where     t.ipaddress=e.ipaddress     and
t.deviceid=g.deviceid   and   t.port_type=:1   and   l.function_name=:2   and
mod(decode(g.functionid,0,l.functionid+1,g.functionid),l.functionid)!=0});



$sth->bind_param(1,$_[5]);
$sth->bind_param(2,$_[6]);
$sth->execute();




  while(@data=$sth->fetchrow_array())
  {
```

```
            my             $ipadd             =             substr($data[0],
index($data[0],"<ipadd>")+7,index($data[0],"</ipadd>")-
index($data[0],"<ipadd>")-7);
            my $current_subnet=GetSubnetID($ipadd,$data[1]);
            $_= $current_subnet;
            $count_distinct_subnet1= $count_distinct_subnet;
            push(@distinct_subnetID, $_) unless ($seen{$_}++);
            $count_distinct_subnet=@distinct_subnetID;

            my @ipdecomp = split(/\./,$ipadd);

                  @ipdecomp=reverse @ipdecomp;

                  my $reverse_ip=join(".",$ipdecomp[0],$ipdecomp[1]);


                  $data[0]=~ s/$ipadd/$reverse_ip/;

                  my $full_line=join("?",$current_subnet,$data[0]);
                  push(@iplist, $full_line);



      $j++;

  }

 if($j==0)
 {
 print "ERROR no ipaddress \n";
 $rescode=-1;
 push(@iplist, $rescode);
 return @iplist;
 }
 else
 {

       my $list_of_subnet=join(":",@distinct_subnetID);
      push(@iplist, $list_of_subnet);
      return @iplist;
 }
}



############subroutine which returns the subnet ID ######################
sub GetSubnetID
{
  my @ip_add = split(/\./,$_[0]);
  my @subnet_ip=split(/\./,$_[1]);
  my $i=0;
  my $temp1='';
  my $bin1='';
  my $temp2='';
  my $bin2='';
  my $binary_ip='';
  my $binary_subnet='';
  my $res1='';
  my $res2='';
```

```
  foreach (@ip_add)
  {
   $temp1=sprintf "%d",$ip_add[$i];

   $bin1=unpack("B32", pack("N", $temp1));

   $temp2=sprintf "%d",$subnet_ip[$i];
   $bin2=unpack("B32", pack("N", $temp2));
   $res1=ProceedSubnetID($bin1,$bin2);
   if($i>0)
     {
       $res2=$res2."\.".$res1;
     }
   else
     {
        $res2=$res1;

     }
   $i++;
  }


  return $res2;
}
#####subroutine which multiply the 2 vectors as .* (Matlab)################
sub ProceedSubnetID
{

  my $l=0;
  my $k=0;
  my $res1=0;
  my $i=0;
  my @bin_add=split(//,$_[0]);
  my @bin_subnet=split(//,$_[1]);
  my $j=@bin_add-1;
  foreach(@bin_add)
    {
      $l=$bin_add[$i]*$bin_subnet[$i];
      $k=sprintf "%d",$l;
      if($k==1)
      {
        $res1=$res1+2**$j;
      }
      $i++;
      $j--;
    }
  return $res1;
}
```

Appendix F The API of the CIC_DB_lib (C code)

```
/************************************************************************
******/
// Author: L.Abadie
// version: v3.2
// CIC_DB_lib API
/************************************************************************
******/

int DBDeconnexion(char* ErrorMessage);

int DBConnexion(char* server,char* usr,char* pwd,char* ErrorMessage);

//******** Get information from CIC DB***************//

int       GetDeviceTypeRow(char*       devitype,int       &len_devtype,char*
devtype_result,char* ErrorMessage); UC 10

int GetDeviceRow_devicename(char* functionaldeviname,int &len_device, char*
device_result,char* ErrorMessage); UC 12, UC13, UC 33

int     GetDeviceRow_devid(int     deviceID,int     &len_device,     char*
device_result,char* ErrorMessage); UC 12, UC13, UC 33

int  GetPortRow_pid(int  portID,  int  &len_port,char*  port_row_result,char*
ErrorMessage); UC 12

int GetPortRow_devname(char* devicename, char* port_nb,int port_way, char*
port_type,int &len_port, char* port_row_result,char* ErrorMessage); UC 12,

int   GetPortRow_devid(int   deviceid,   char*   port_nb,int   port_way,char*
port_type,int &len_port, char* port_row_result,char* ErrorMessage); UC 12,

int    GetMacroConnectivityRow_lkid(int    lkID,    int    &len_conn,char*
Conn_row,char* ErrorMessage); UC 7, UC 15

int   GetMacroConnectivityRow_node(int   nodeID,   char*   port_nb,   int
port_way,char* port_type,int &len_conn, char* Conn_row,char* ErrorMessage);
UC 7, UC 15

int  GetMacroConnectivityRow_nodename(char*  node_name,  char*  port_nb,  int
port_way,char* port_type,int &len_conn, char* Conn_row,char* ErrorMessage);
UC 7, UC 15

int   GetIPInfoRow(char*   ip_address,int   &len_ip,   char*     IP_row,char*
ErrorMessage); UC 12

int     GetLkTypeRow_lkname(char*     lktype_name,int     &len_lktype,char*
LkType_row,char* ErrorMessage); UC 30

int GetLkTypeRow_lknb(int lktype_nbr,int &len_lktype,char* LkType_row,char*
ErrorMessage); UC 30

int   GetLkTypeDecomposition_lknb(int   lktype_nbr,int   &len_array,char*
LkType_row,char* ErrorMessage); UC 30
```

```
int GetDeviceNamesPerType(char* devitype, int &len_array, char*
devIDs_list,char* ErrorMessage); UC 18

int GetDeviceIDsPerType(char* devitype, int &len_array, int*
devIDs_list,char* ErrorMessage); UC 18

int GetLkFromDevID(int node_from, int &len_array, int* lkfrom_list,char*
ErrorMessage); UC 18

int GetLkToDevID(int node_to, int &len_array, int* lkfrom_list,char*
ErrorMessage); UC 18

int GetPortIDPerDevID(int devID, int &len_array, int* portID_list,char*
ErrorMessage); UC 12

int GetMacIPAddPerDevID(int devID, int &len_mac,char* MacAdd_list,char*
ErrorMessage); UC 12

int GetDestinationNamePerDevPort(char* devname,char* port_nb, char*
port_type,int &len_Destin_list,char* Destin_list,char* ErrorMessage); UC 19

int GetFreeDeviceNamesPerType(char* devitype, int &len_array, char*
devIDs_list,char* ErrorMessage); UC 18,

int GetDetailedConnectivityBetweenDevices(char* dev_from, char* dev_to,
char* nfrom_list,int& len_array,int & nfrom_arraylen,int & pfrom_arraylen,
int & nto_arraylen, int & pto_arraylen,int &lkinfo_arraylen,char*
pfrom_list,char* pto_list,int* pfrom1_list,int* pto1_list,char*
nto_list,int* pathid_list, int* link_pos_list,char* lkinfo_list,int
reload_connectivity,int delete_connectivity,char* ErrMess); UC 6, UC 30

int GetDetailedConnBetweenDeviceDevType(char* dev_from, char* dev_to,int
devto_type_given, char* nfrom_list,char* pfrom_list,char* pto_list,int&
len_array,int & nfrom_arraylen,int & pfrom_arraylen, int & nto_arraylen,
int & pto_arraylen,int &lkinfo_arraylen, int* pfrom1_list,int*
pto1_list,char* nto_list,int* pathid_list, int* link_pos_list,char*
lkinfo_list,int reload_connectivity,int delete_connectivity,char* ErrMess);
UC 6, UC 32

int GetConnectivityBetweenDevices(char* dev_from, char* dev_to, int&
len_array, int &pfrom1_arraylen,int &nfrom1_arraylen,int &pto1_arraylen,int
&nto1_arraylen,int &lkinfo_arraylen,char* nfrom1_list,char* nto1_list,char*
pfrom1_list,int* pfrom_list,char* pto1_list,int* pto_list,int*
pathid_list,int* link_pos_list,char* lkinfo_list,int
reload_connectivity,int delete_connectivity,char* ErrMess); UC 6, UC 32

int GetConnOutBetweenDeviceDeviceType(char* dev_from, char* devtype,int
devtype_give, int& len_array, char* pfrom_arraylist, int
reload_connectivity,int delete_connectivity,char* ErrMess); UC 6

int GetBusySubsystems(int &len_array1,char* ListSubsystemUsed,char*
ListSubsystemUsed1,int &len_array, int* ListSubsystemState, char*
ErrorMessage); UC 17

int GetDeviceNamesPerLocation(char* location, int &len_array, char*
devnames_list,char* ErrorMessage); UC 19, UC 33

int GetSpareHWPerLocation(char* location, int &len_array, char*
devnames_list,char* ErrorMessage); UC 20, UC 22
```

```
int     GetSpareHWPerType(char*     hwtype,     int     &len_array,     char*
devnames_list,char* ErrorMessage); UC 20, UC 22

int  GetFunctionalDeviceStatus(char*  devicename,char*  device_status  ,char*
ErrMess); UC 24

int  GetHWDeviceStatus(char*  serialnb,char*  device_status  ,char*  ErrMess);
UC 25

int    GetHWDeviceRow_serialnb(char*    serialnb,int    &len_device,    char*
device_result,char* ErrorMessage); UC 20

int GetFunctionalDeviceByStatus(char* system_name, char* device_status, int
&len_devlist , char* device_list,char* ErrMess); UC 30

int              GetHWLastFunctionalDevice(char*              serialnb,char*
functional_devicename,int &len_functionaldname ,char* ErrMess); UC 21

int       GetFunctionalDeviceLastHW(char*       functional_devicename,char*
serialnb,int &len_serialnb ,char* ErrMess); UC 29

int  GetHistoryOfHWDevice(char*  serialnb,char*  functionaldevice_history,int
&len_history , char* min_date, char* max_date,char* ErrMess); UC 22

int    GetHWDeviceByStatus(char*    system_name,    char*    device_status,int
&len_status ,char* functionaldevice_status,char* ErrMess); UC 20, UC 21

int      GetHistoryOfFunctionalDevice(char*      functional_devicename,char*
functionaldevice_history,int    &len_history    ,    char*    min_date,    char*
max_date,char* ErrMess); UC 22, UC 33

int GetBoardCpntRow_cpntname(char* functionalcpntname,int &len_cpnt, char*
cpnt_result,char* ErrorMessage); UC 22

int      GetBoardCpntRow_cpntid(int      cpntID,int      &len_cpnt,      char*
cpnt_result,char* ErrorMessage); UC 22

int     GetMicroConnectivityRow_lkid(int     lkID,     int     &len_conn,char*
Conn_row,char* ErrorMessage); UC 28

int GetMicroConnectivityRow_node(int cpntID, int port_nb, int port_way,int
&len_conn, char* Conn_row,char* ErrorMessage); UC 28

int  GetMicroConnectivityRow_cpntname(char*  cpnt_name,  int  port_nb,  int
port_way,int &len_conn, char* Conn_row,char* ErrorMessage); UC 28

int    GetBoardCpntPerType(char*    cpnttype,    int    &len_array,    char*
cpntIDs_list,char* ErrorMessage); UC 32

int GetMicroLkFromCpntID(int cpntid_from,int motherboardID, int &len_array,
int* lkfrom_list,char* ErrorMessage); UC 28

int GetMicroLkToCpntID(int cpnt_to,int motherboardID, int &len_array, int*
lkfrom_list,char* ErrorMessage); UC 28

int  GetCpntNamesPerBoard(char*  motherboardname,  int  &len_array,  char*
devnames_list,char* ErrorMessage); UC 22

int  GetSpareHWCpntPerLocation(char*  location,  int  &len_array,  char*
devnames_list,char* ErrorMessage); UC 20
```

```
int    GetSpareHWCpntPerType(char*    hwtype,    int    &len_array,    char*
devnames_list,char* ErrorMessage);UC 20, UC 28

int    GetHWCpntRow_serialnb(char*    serialnb,int    &len_device,    char*
device_result,char* ErrorMessage);UC 20

int GetHWCpntRow_snbid(int snbid,int &len_device, char* device_result,char*
ErrorMessage);UC 20

int  GetFunctionalBoardCpntStatus(char*  cpntname,char*  cpnt_status  ,char*
ErrMess); UC 28

int GetHWBoardCpntStatus(char* serialnb,int snbid,char* cpnt_status ,char*
ErrMess); UC 29

int    GetHWLastFunctionalBoardCpntName(char*    serialnb,int    snbid,char*
functional_cpntname,int &len_functionaldname ,char* ErrMess); UC 21

int     GetFunctionalBoardCpntNameLastHW(char*     functional_cpntname,char*
serialnb,int &len_serialnb ,char* ErrMess);UC 29

int    GetHistoryOfFunctionalBoardCpntName(char*    functional_cpntname,char*
functionalcpnt_history,int    &len_history    ,    char*    min_date,    char*
max_date,char* ErrMess); UC 22, UC 30

int       GetHistoryOfHWBoardCpnt(char*       serialnb,int       snbid,char*
functionalcpnt_history,int    &len_history    ,    char*    min_date,    char*
max_date,char* ErrMess); UC 22, UC 30

int              GetHWBoardCpntByStatus(char*              cpnt_status,char*
functionalcpnt_status,int &len_status , char* ErrMess); UC 20,UC 21

int GetMicroConnectFromPortid(int mboardportid_from, int mboardportid_to,
char* cpntname,int* pathid_list, int* link_pos_list,int &len_array,char*
nfrom_list,int    &nfrom_arraylen,char*    nto_list,int    &nto_arraylen,char*
last_nfrominfo,int         &len_lastnfrominfo,char*         last_ntoinfo,int
&len_lastntoinfo,char* ErrMess); UC 32,

int GetMicroConnectToCpntType(int mboardportid_from, int cpnttype_given,
char* cpnttype,int* pathid_list, int* link_pos_list,int &len_array,char*
nfrom_list,int    &nfrom_arraylen,char*    nto_list,int    &nto_arraylen,char*
last_nfrominfo,int         &len_lastnfrominfo,char*         last_ntoinfo,int
&len_lastntoinfo,char* ErrMess); UC 32

int GetMicroConnectBetweenBoardCpntAndMotherBoard(char* cpntname_from, int
mboard_portid, int* pathid_list, int* link_pos_list,int &len_array,char*
nfrom_list,int    &nfrom_arraylen,char*    nto_list,int    &nto_arraylen,char*
last_nfrominfo,int         &len_lastnfrominfo,char*         last_ntoinfo,int
&len_lastntoinfo,char* ErrMess); UC 32

int   GetPortPerSubsystem(char*   dev_from,   char*   subsystem_name,   int&
len_array, char* pfrom_list, char* ErrMess); UC 17,

int    GetIPAliasesPerIPName(char*    ipname,    int    &len_array,    char*
ipaliases_list,char* ErrorMessage); UC 12, UC 14

int GetIPAliasRow(char* ipalias,int &len_ipalias,char* ipalias_result,char*
ErrorMessage); UC 12,

int GetLkIDsPerLkInfo(char* lkinfo, int &len_array, int* lkIDs_list,char*
ErrorMessage); UC 7
```

```
int GetDeviceNamesPerFunction(char* function, int &len_array, char*
devnames_list,char* ErrorMessage); UC 13

int GetBootImageRow_devicename(char* functionaldeviname,int &len_device,
char* device_result,char* ErrorMessage); UC 12

int LoadRoutingtable( char* switch_name,int &len_array_destinname,char*
destinname_list, int &len_array_destinip,char* destinip_list, int
&len_array_portnext,char* port_list,int &len_array_ipaddnext,char*
ipaddnext_list,int &len_array_subnetnext,char* subnetnext_list,int
&len_array_macaddnext,char* macaddnext_list,char* ErrMess); UC 8

int CreateDestinationTable(char* devicename, int round_trip_max,char*
ErrMess); UC 9

int CreateRoutingTable(char* devicename, int round_trip_max,char* ErrMess);
UC 8

/****Internal functions for CDBVis to improve performances***************/

int LoadLinkTypeTable(int &len_lkid, int* lkid_list,int& len_lkname,char*
lktypename_list,char* ErrMess);

int LoadDeviceTypeTable(char* systemnameList,int &len_did, char*
devtypename_list,int &len_array,int* devtypeID_list, char* ErrMess);

int MatchDeviceIDDeviceName(char* systemnameList,int &len_did, int*
devid_list,int& len_dname,char* devname_list,char* ErrMess);

int LoadConnectivityTable(char* systemnameList,int &len_array, int*
lkid_list, int * node_from_list,int* node_to_list,int &portfrom_len,int
&portto_len,char* port_nbrfrom_list, char* port_nbrto_list, int*
bidirectional_list, int* lkused_list, int* lktype_list,char*
lkinfo_list,int & lkinfo_len,char* ErrMess);

int GetAvailableFunctions(int &len_array, char* function_list,char*
ErrorMessage);

int DropTableSchema(char* ErrMess); //should not be used

int CreateTableSchema(char* ErrMess); // only used once

int GetListOfSubsystems(int &len_array, char* sysname_list,char*
ErrorMessage);

int GetAllPathsPerDevice(char* systemnameList,char* devname,int& len_array,
int* lkid_list5,int* pathid_list,int* link_pos_list,int
reload_connectivity,int delete_connectivity,char* ErrMessage);

int GetDeviceID_devicename(char* deviname,int &deviceID,char*
ErrorMessage);

int GetDeviceName_deviceid(int deviceID,char* devicename,char*
ErrorMessage);

int GetPortID_portinfo(int deviceID,char* port_nb,char* port_type,int
port_way,int &portID,char* ErrorMessage);

int GetCpntID_cpntname(char* cpntname,int &cpntID,char* ErrorMessage);
```

```
int GetCpntName_cpntid(int cpntID,char* cpntname,char* ErrorMessage);

/*************** insert functions ***************/

int    InsertPort(char*    devicename,char*    port_nb,int    admin_status,int
port_way,int    speed,int    pxi_booting,char*    port_type,char*    bia,char*
ipadd1,char* ipname,char* subnet,char* macadd,char* phy,int last_rows,char*
ErrMess); UC 8, UC 11, UC 16

int  InsertCompositeLinkType(  char*  link_name,char  *  simple_lk_list,int
last_rows,char* ErrMess); UC 8, UC 16, UC 16

int  InsertSimpleLinkType(char  *link_name,int  last_rows,char*  ErrMess);  UC
8, UC 16

int  InsertMacroLink(char*  node_from,char*  node_to,char*  port_nbfrom,char*
port_nbto,char*    port_typefrom,char*    port_typeto,char*    link_type,char*
link_information,int     bidirectional_link_used,     int     last_rows,char*
ErrMess);UC 8, UC 11, UC 16

int    InsertFunctionalDevice(char*    systemnameList,char*    devicename,char*
devicetype,int node,int promiscuous_mode,char* serial_nb,char* hwtype,char*
responsible,char*     comments,char*     location,char*     function_list,int
last_rows,char* ErrMess); UC 8, UC 11, UC 16

int      InsertDeviceType(char*      systemnameList,char*      devicetype,int
nbrofinput,int    nbrofoutput,    char*    description,    char*    rgbcolor,int
last_rows,char* ErrMess); UC 8, UC 11

int InsertMultiplePorts(char* devicename,char* port_nb,int admin_status,int
port_way,int    speed,int    pxi_booting,char*    port_type,char*    bia,char*
ipadd1,char* ipname,char* subnet,char* macadd,char* phy,int first_time1,int
last_rows1,char* ErrMess); UC 8, UC 11, UC 16

int   InsertMultipleSimpleLinkTypes(   char   *link_name,int   first_time1,int
last_rows1,char* ErrMess); UC 8, UC 16

int     InsertMultipleCompositeLinkTypes(     char*     link_name,char     *
simple_lk_list,int first_time1,int last_rows1,char* ErrMess); UC 8, UC 16

int   InsertMultipleDeviceTypes(char*   systemnameList,char*   devicetype,int
nbrofinput,int    nbrofoutput,    char*    description,char*    rgbcolor,int
first_time1, int last_rows1,char* ErrMess); UC 8, UC 11, UC 16

int        InsertMultipleFunctionalDevices(char*        systemnameList,char*
devicename,char*     devicetype,int     node,int     promiscuous_mode,char*
serial_nb,char*      hwtype,char*      responsible,char*      comments,char*
location,char* function_list,int first_time1,int last_rows1,char* ErrMess);
UC 8, UC 11, UC 16

int      InsertMultipleSpareDevices(char*      hwname,char*      hwtype,char*
serial_nb,char*      responsible,char*      comments,char*      location,int
first_time1,int last_rows1,char* ErrMess); UC 20,

int   InsertSpareDevice(char*   hwname,char*   hwtype,char*   serial_nb,char*
responsible,char* comments,char* location,int last_rows1,char* ErrMess); UC
20

int     InsertMultipleMacroLinks(char*     node_from,char*     node_to,char*
port_nbfrom,char*   port_nbto,char*   port_typefrom,char*   port_typeto,char*
```

```
link_type,char*       link_information,int       bidirectional_link_used,int
first_time1, int last_rows1,char* ErrMess); UC 8, UC 16

int      InsertMultipleSparePorts(char*      serialnb,char*      port_nb,int
port_way,char*   port_type,char*   bia,char*   macadd,int   first_time1,int
last_rows1,char* ErrMess); UC 20

int   InsertSparePort(char*   serialnb,char*   port_nb,int   port_way,char*
port_type,char* bia,char* macadd,int last_rows,char* ErrMess); UC 20

int   InsertTestFunctionalDevice(char*   location,char*   ipaddressList,int
ipaddlen,char*      ipnameList,int      ipnamelen,char*      subnetList,int
subnetlen,char*      portnbList,int      portnblen,char*      portypeList,int
ptypelen,int* portwayList,int nb_ipadd,int last_rows,char* ErrMess); UC 20

int   InsertMultipleBootImages(char*   devicename,char*   boot_image,char*
kernel_image_location,char*                    initrd_image_location,char*
physical_location,char*          boot_protocol,int     first_time1,     int
last_rows1,char* ErrMess); UC 8, UC 11, UC 16

int      InsertMultipleBoardCpnts(char*      cpntname,char*      cpnttype,int
replacable,char*    motherboardname,char*    serial_nb,char*    hwtype,char*
responsible,char*      comments,char*      location,int      first_time1,int
last_rows1,char* ErrMess); UC 28, UC 32, UC 34

int      InsertMultipleMicroLinks(char*      node_from,char*      node_to,int
port_nbfrom,int  port_nbto,char*  link_type,int  bidirectional_link_used,int
first_time1, int last_rows1,char* ErrMess); UC 28, UC 32

int InsertSubsystem(char *system_name,char* parent_sysname,char* ErrMess);
UC 31, UC 32, UC 33, UC 34

int InsertMultipleIPAliases(char* ipaddress,char* ipname,char* ipalias,int
first_time1,int last_rows1,char* ErrMess); UC 8, UC 11, UC 16

int   InsertSimpleDeviceFunction(char   *function_name,int   last_rows,char*
ErrMess); UC 8, UC 16

/*************** update functions for inventory/history ***************/

int ReplaceFunctionalDevice(char* devicename,char* new_device_status,char*
new_location,char*       user_comments,char*       status_datechange,char*
serialnb_replaced,char* replace_date,char* ErrMess); UC 23, UC 24, UC 25,
UC 29,

int    SetToTestUseStatus(char*    devicename,char*    user_comments,char*
status_change,char*      serialnb_replaced,char*      testboard_name,char*
replace_date,char* ErrMess); UC 26, UC 34

int ReplaceFunctionalBoardCpnt(char* cpntname,char* new_cpnt_status,char*
new_location,char*       user_comments,char*       status_datechange,char*
serialnb_replace,char* replace_date,char* ErrMess); UC 28

int            SwapTwoDevices(char*            functional_devicename1,char*
functional_devicename2,char* comments,char* ErrMess); UC 33

int   UpdateHWDeviceStatus(char*   serialnb,char*   new_device_status,char*
new_location,char*       user_comments,char*       status_datechange,char*
functional_devicename,char* ErrMess); UC 23, UC 34
```

```
int   UpdateHWBoardCpntStatus(char*   serialnb,char*   new_cpnt_status,char*
new_location,char*        user_comments,char*        status_datechange,char*
functional_cpntname,char* ErrMess); UC 28, UC 29

int     UpdateMultiplePortStatuses(char*     devicename,char*     port_nb,int
port_way,char*      port_type,int      admin_status,int      first_time1,int
last_rows1,char* ErrMess); UC 12

int  UpdateMultipleDeviceNodeUsedByType(char*  devicetype,int  nodeused,int
devtype_given,int last_rows,char* ErrMess); UC 10

int     UpdateMultipleDeviceNodeUsed(char*     devicename,int     nodeused,int
first_time1,int last_rows1,char* ErrMess); UC 10

int     UpdateMultipleLkUsedLinks(char*     node_from,char*     node_to,char*
port_nbrfrom,char*   port_nbrto,char*   port_typefrom,char*   port_typeto,int
lkused,int first_time1, int last_rows1,char* ErrMess); UC 13

/*************** delete functions ***************/

int DeleteIPAlias(char* ipalias,char* ErrMess); UC 16

int DeleteBootImage(char* devicename,char* ErrMess); UC 16

int DeleteHWDevice(char* serialnb,char* ErrMess); UC 16

int DeleteFunctionalDeviceType(char* devicetype,char* ErrMess); UC 16

int DeleteFunctionalDevice(int deviceid,char* ErrMess); UC 16

int DeleteLinkRow(int linkid,int macro_link,char* ErrorMessage); UC 16

int DeletePortRow(int portid,char* ErrMess); UC 16

int DeleteSystemName(char* sysname,char* ErrMess);

/*************update functions in case of mistype***************/
int   UpdateSubsystemName(char*   old_systemname,char*   new_systemname,char*
ErrMess);

int  UpdateMultiplePorts(char*  devicename,char*  port_nb,int  port_way,char*
port_type,int    speed,char*    phy,int    pxi_booting,int    first_time1,int
last_rows1,char* ErrMess);

int   UpdateMultipleAttributeMacIPs(char*   ip_add,char*   subnet_mask,char*
ipname,int first_time1,int last_rows1,char* ErrMess);

int    UpdateMultipleIPAddresses(    char*    ip_add,char*    oldip_add,int
first_time1,int last_rows1,char* ErrMess);

int      UpdateMultipleCompositeLinkTypes(char*      link_name,char      *
simple_lk_list,int first_time1,int last_rows1,char* ErrMess);

int  UpdateMultipleLinkTypeNames(char  *old_link_name,  char  *link_name,int
first_time1,int last_rows1,char* ErrMess);

int     UpdateMultipleLkInfoLinks(char*     node_from,char*     node_to,char*
port_nbrfrom,char*  port_nbrto,char*  port_typefrom,char*  port_typeto,char*
link_information,int first_time1, int last_rows1,char* ErrMess);
```

```
int   UpdateMultipleDeviceTypes(char*   devicetype_old,char*   devicetype,int
first_time1,int last_rows1,char* ErrMess);

int        UpdateMultipleDeviceTypeAttributes(char*        devicetype,char*
description,int      nbrofinput,int      nbrofoutput,char*      rgbcolor,int
first_time1,int last_rows1,char* ErrMess);

int    UpdateMultipleDTypeDevices(char*    devicename,char*    devicetype,int
first_time1,int last_rows1,char* ErrMess);

int UpdateMultipleDevNamesDevices(char* devicename_old,char* devicename,int
first_time1,int last_rows1,char* ErrMess);

int     UpdateMultipleDeviceActive(char*     devicename,int     active,int
first_time1,int last_rows1,char* ErrMess);

int    UpdateMultipleDeviceActiveByType(char*    devicetype,int    active,int
devtype_given,int last_rows,char* ErrMess);

int    UpdateMultipleAttributesDevices(char*    devicename,int    node,int
promiscuous_mode,char*   location,char*   function_list,int   first_time1,int
last_rows1,char* ErrMess); UC 10, UC 27

int    UpdateMultipleLkTypeLinks(char*    node_from,char*    node_to,char*
port_nbrfrom,char*  port_nbrto,char*  port_typefrom,char*  port_typeto,char*
link_type,int first_time1, int last_rows1,char* ErrMess);

int  UpdateMultipleBidirectionalLinks(char*  node_from,char*  node_to,char*
port_nbrfrom,char*   port_typefrom,char*   port_nbrto,char*   port_typeto,int
bidirectional_link_used,int first_time1, int last_rows1,char* ErrMess);

int          UpdateDeviceFunctionName(char*          old_function_name,char*
new_function_name,char* ErrMess);

int UpdateHWSerialNB(char* old_serialnb,char* new_serialnb,char* ErrMess);

int UpdateIPalias(char* old_ipalias,char* new_ipalias,char* ErrMess);

int        UpdateMultipleDeviceTypeSystemList(char*        devicetype,char*
new_systemList,int first_time1,int last_rows1,char* ErrMess);

int       UpdateBootImage(char*       devicename,char*       boot_image,char*
kernel_image_location,char*                    initrd_image_location,char*
physical_location,char*  boot_protocol,char* ErrMess);

int          UpdateMultipleDeviceSystemList(char*          devicename,char*
new_systemList,int first_time1,int last_rows1,char* ErrMess);
```

Appendix G Example of a select query using OCI

To perform a SQL statement, the following common steps (quite similar to JDBC, ODBC or PERL DBI) are required:

- use of **OCIHandleAlloc** to allocate a handle.
- use of **OCIStmtPrepare** to prepare the query. A query can consists of a block of statements or a call to PL/SQL functions/procedures.
- use of **OCIBindByName** if there are some bind variables;
- use of **OCIStmtExecute** to execute the statement;
- use of **OCIDefineByPos** to define where to put the content of a column. To be used only for SELECT statement (not used in Perl DBI or ODBC);
- use of **OCIStmtFetch2** to fetch the results of the statement (only for SELECT statement).

There is an example of a C code below. It returns the row of a given devicetype. Functions provided by OCI are in bold.

```
/*************************************************************************
******/
// Author: L.Abadie
// Code compatible with version: v3.2 of the confDB.lib
//    Get    the    row    associated    with    the    given    device    type
/*************************************************************************
******/

// allocate the handle (memory)
status =OCIHandleAlloc (ociEnv, (void**)&stmthp, OCI_HTYPE_STMT , 0, 0);
if(status!=OCI_SUCCESS)
{
       rescode=ShowErrors (status, ociError, "OCIStmtHAndleStmt unsuccessful");
       if(ociError!=0)
              OCIReportError(ociError,appliName,ErrorMessage,1);
       else
              GetErrorMess(appliName,"NOT CONNECTED TO ANY DB",ErrorMessage,1);
       return -1;
}
else
{
       char selectdevtype[1000];
       //write the select statement
       sprintf(selectdevtype,"select     devicetype,devicetypeid,description,rgbcolor
from %s where devicetype=:dtype ",DEVICETYPE_TABLE);
       //prepare the statement
       status=OCIStmtPrepare(stmthp,     ociError,     (text*)     selectdevtype,(ub4)
strlen(selectdevtype),(ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
}

if(status!=OCI_SUCCESS )
{
       if(rescode==0)
       {
              rescode=ShowErrors (status, ociError, "OCIStmtPrepare unsuccessful");
       }
}
else
       //make the binding variable
       status=OCIBindByName(stmthp,    &bnd1p,    ociError,(text*)":dtype",-1,(dvoid*)
devitype,strlen(devitype)+1,  SQLT_STR,  (dvoid  *)  0,(ub2  *)  0,  (ub2*)  0,  (ub4)  0,
(ub4 *) 0, OCI_DEFAULT);

if(status!=OCI_SUCCESS)
{
       if(rescode==0)
```

```
        {
                rescode=ShowErrors (status, ociError, "OCIBindByPos unsuccessful");
        }
}
else
        //execute the statement
        status=OCIStmtExecute(ociHdbc, stmthp, ociError, 0, 0,(OCISnapshot *) 0,
(OCISnapshot *) 0, OCI_DEFAULT);

if(status!=OCI_SUCCESS)
{
        if(rescode==0)
        {

                rescode=ShowErrors (status, ociError, "OCIAttrGet2 unsuccessful");
        }
}
else
{
                //define the column to associate each of them with a C variable
                status   =OCIDefineByPos(stmthp,   &def[0],   ociError,1,   (ub1   *)
(devicetype_temp), devtypelen + 1,SQLT_STR, (dvoid *)&dtype_null,(ub2 *) 0,0,
OCI_DEFAULT);
}
….

if(status!=OCI_SUCCESS)
{
        if(rescode==0)
        {
                rescode=ShowErrors (status, ociError, "OCIDefineByPos4 unsuccessful");
        }
}
else   //fetch the result : we get one row one by one
        status =OCIStmtFetch2(stmthp,ociError, 1, OCI_FETCH_NEXT,1, OCI_DEFAULT);
```

In parallel, there are other OCI functions to tune SQL statements such as the prefetch size memory by using **OCIAttrSet** (stmthp, OCI_HTYPE_STMT, &prefetch_rows, 0, **OCI_ATTR_PREFETCH_ROWS**, ociError). Prefetch_rows sets the number of rows which should be returned in one round trip. The typical value is between 1000 and 10000. It corresponds to the number of round trip (between the client and the DB server) and it should be set according to the network load average and the type of query. For instance, if a query will never return more than 1000 rows, there is no point in setting this parameter to 10,000. The library can be used on Linux and Windows.

Appendix H The PVSS CIC_DB_lib interface

```
/******************************************************************************
******/
// Author: L.Abadie
// Code compatible with version: v3.2 of the CIC_DB_lib
//PVSS interface
/******************************************************************************
******/

/*****************************************************************************/
BaseExternHdl* newExternHdl(BaseExternHdl* nextHdl){
      static FunctionListRecEx funcs[]={

FunctionListRecEx(TEXT_VAR,          "PVSSGetDeviceTypeRow",          "(string
dtype)",true,reinterpret_cast<void             (*)()>(PVSSGetDeviceTypeRow),
TextVar(0)),
FunctionListRecEx(INTEGER_VAR,         "PVSSDBDeconnexion",          "(string
ErrMess)",true,reinterpret_cast<void              (*)()>(PVSSDBDeconnexion),
IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,  "PVSSDBConnexion",  "(string  server,string
usr,string     pwd,string              ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSDBConnexion), IntegerVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSGetDeviceRow_devicename",      "(string
dname)",true,reinterpret_cast<void      (*)()>(PVSSGetDeviceRow_devicename),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,        "PVSSGetDeviceName_deviceid",        "(int
deviceid)",true,reinterpret_cast<void    (*)()>(PVSSGetDeviceName_deviceid),
TextVar(0)),
FunctionListRecEx(INTEGER_VAR,     "PVSSGetDeviceID_devicename",     "(string
devicename,string                       ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetDeviceID_devicename), TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSGetDeviceRow_devid",          "(int
dID)",true,reinterpret_cast<void             (*)()>(PVSSGetDeviceRow_devid),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,  "PVSSGetPortRow_devid",  "(int  devID,  string
pt_nb,int       pway,       string       ptype)",true,reinterpret_cast<void
(*)()>(PVSSGetPortRow_devid), TextVar(0)),
FunctionListRecEx(TEXT_VAR,            "PVSSGetPortRow_pid",            "(int
dID)",true,reinterpret_cast<void (*)()>(PVSSGetPortRow_pid), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetMacroConnectivityRow_node",     "(int
nodeID,  string  ptnb,int  pway,string  ptype)",true,reinterpret_cast<void
(*)()>(PVSSGetMacroConnectivityRow_node), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetMacroConnectivityRow_lkid",     "(int
lkID)",true,reinterpret_cast<void  (*)()>(PVSSGetMacroConnectivityRow_lkid),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,           "PVSSGetIPInfoRow",           "(string
ip_add)",true,reinterpret_cast<void (*)()>(PVSSGetIPInfoRow), TextVar(0)),
FunctionListRecEx(TEXT_VAR,        "PVSSGetLkTypeRow_lkname",        "(string
lkname)",true,reinterpret_cast<void         (*)()>(PVSSGetLkTypeRow_lkname),
TextVar(0)),
FunctionListRecEx(DYN_VAR,           "PVSSGetDeviceIDsPerType",      "(string
dtype,string                            ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetDeviceIDsPerType), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,       "PVSSGetDeviceNamesPerType",    "(string
dtype,dyn_string    devnames,dyn_int    devids)",true,reinterpret_cast<void
(*)()>(PVSSGetDeviceNamesPerType), IntegerVar(0)),
```

```
FunctionListRecEx(DYN_VAR,      "PVSSGetFreeDeviceNamesPerType",    "(string
dtype)",true,reinterpret_cast<void    (*)()>(PVSSGetFreeDeviceNamesPerType),
TextVar(0)),
FunctionListRecEx(DYN_VAR,     "PVSSGetDestinationNamePerDevPort",   "(string
devname,string        ptnb,string        ptype)",true,reinterpret_cast<void
(*)()>(PVSSGetDestinationNamePerDevPort), TextVar(0)),
FunctionListRecEx(DYN_VAR,    "PVSSGetLkFromDevID",   "(int  node_from,string
ErrMess)",true,reinterpret_cast<void            (*)()>(PVSSGetLkFromDevID),
IntegerVar(0)),
FunctionListRecEx(DYN_VAR,      "PVSSGetLkToDevID",    "(int   node_to,string
ErrMess)",true,reinterpret_cast<void              (*)()>(PVSSGetLkToDevID),
IntegerVar(0)),
FunctionListRecEx(DYN_VAR,     "PVSSGetPortIDPerDevID",   "(int  devid,string
ErrMess)",true,reinterpret_cast<void     (*)()>(     PVSSGetPortIDPerDevID),
IntegerVar(0)),
FunctionListRecEx(DYN_VAR,          "PVSSGetMacIPAddPerDevID",          "(int
devid)",true,reinterpret_cast<void     (*)()>(     PVSSGetMacIPAddPerDevID),
IntegerVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSGetLkTypeRow_lknb",          "(int
lktype_nb)",true,reinterpret_cast<void        (*)()>(PVSSGetLkTypeRow_lknb),
TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSInsertDeviceType", "(string sysname,string
dtype,int   nbinput,int   nboutput,string   description,string   rgbcolor,int
last_rows)",true,reinterpret_cast<void         (*)()>(PVSSInsertDeviceType),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSInsertFunctionalDevice",      "(string
sysname,string   dname,string   dtype,int   node,int   promiscuous_mode,string
serialnb,    string    hwtype,string    responsible,string    location,string
comments,string   function_list,int   last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertFunctionalDevice), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSInsertPort",    "(string    dname,string
pt_nb,string pt_type,int pway,string bia,string ipname,string ipadd,string
subnet,string    macadd,int    pxibooting,int    speed,    string    phy,int
last_rows)",true,reinterpret_cast<void (*)()>(PVSSInsertPort), TextVar(0)),
FunctionListRecEx(TEXT_VAR,       "PVSSInsertSimpleLinkType",       "(string
lktype_name,int                    last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertSimpleLinkType), TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSInsertMacroLink", "(string sysname,string
node_from,string    node_to,string    port_from,int    port_typefrom,string
port_to,int       port_typeto       ,string       link_type_name       ,int
bidirectional_link,string                                    link_info,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMacroLink),TextVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSInsertCompositeLinkType",      "(string
sysname,string   lktype_name,dyn_string   lktype_list,int   first_time,int
last_rows)",true,reinterpret_cast<void  (*)()>(PVSSInsertCompositeLinkType),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSInsertMultipleDeviceTypes",     "(string
sysname,string  dtype,int  nbinput,int  nboutput,string  description,string
rgbcolor,int      first_time,int      last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleDeviceTypes), TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSInsertMultipleFunctionalDevices", "(string
sysname,string   dname,string   dtype,int   node,int   promiscuous_mode,string
serialnb,    string    hwtype,string    responsible,string    location,string
comments,string               function_list,int               first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleFunctionalDevices), TextVar(0)),
FunctionListRecEx(TEXT_VAR,       "PVSSInsertMultiplePorts",       "(string
dname,string pt_nb,string pt_type,int pway,string bia,string ipname,string
ipadd,string subnet,string macadd,int pxibooting,int speed, string phy,int
```

```
first_time,int                          last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultiplePorts), TextVar(0)),
FunctionListRecEx(TEXT_VAR,  "PVSSInsertMultipleSimpleLinkTypes",  "(string
lktype_name,int    first_time,int    last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleSimpleLinkTypes), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSInsertMultipleMacroLinks",    "(string
node_from,string    node_to,string    port_from,int    port_typefrom,string
port_to,int         port_typeto,string         link_type_name         ,int
bidirectional_link,string           link_info,int           first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleMacroLinks),TextVar(0)),
FunctionListRecEx(TEXT_VAR,         "PVSSInsertMultipleCompositeLinkTypes",
"(string     lktype_name,dyn_string     lktype_list,int     first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleCompositeLinkTypes), TextVar(0)),
FunctionListRecEx(INTEGER_VAR,             "PVSSCreateTableSchema","(string
ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSCreateTableSchema),IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,               "PVSSDropTableSchema","(string
ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSDropTableSchema),IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,            "PVSSCreateRoutingTable","(string
devicename,int   roundtrip,string       ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSCreateRoutingTable),IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,        "PVSSCreateDestinationTable","(string
devicename,int   roundtrip,string       ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSCreateDestinationTable),IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,  "PVSSGetConnectivityBetweenDevices","(string
dfrom,  string  dto,dyn_string  nfrom_list,  dyn_string  pfrom_list,dyn_int
pwayfrom_list,dyn_string     nto_list,dyn_string     pto_list,     dyn_int
pwayto_list,     dyn_int               pid_list,dyn_string     link_info,int
reload_connectivity,delete_connectivity,string
ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetConnectivityBetweenDevices), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSGetConnOutBetweenDeviceDeviceType","(string   dfrom,   string   dto,int
devtype_given,   dyn_string        pfrom_list,int   reload_connectivity,int
delete_connectivity,string             ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetConnOutBetweenDeviceDeviceType), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSGetDetailedConnBetweenDeviceDevType","(string  dfrom,  string  dto,int
dto_typegiven,dyn_string         nfrom_list,dyn_string   pfrom_list,dyn_int
pwayfrom_list,  dyn_string  nto_list,  dyn_string       pto_list,  dyn_int
pwayto_list,  dyn_int       pid_list,  dyn_int       lkpos_list,dyn_string
link_info,int       reload_connectivity,int       delete_connectivity,string
ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetDetailedConnBetweenDeviceDevType), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSGetDetailedConnBetweenDeviceDevType_plsql","(string    dfrom,    string
dto,int dto_typegiven,dyn_string   nfrom_list,dyn_string pfrom_list,dyn_int
pwayfrom_list,  dyn_string  nto_list,  dyn_string       pto_list,  dyn_int
pwayto_list,  dyn_int       pid_list,  dyn_int       lkpos_list,dyn_string
link_info,string                        ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetDetailedConnBetweenDeviceDevType_plsql), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSGetDetailedConnectivityBetweenDevices","(string     dfrom,     string
dto,dyn_string          nfrom_list,    dyn_string    nto_list,    dyn_string
pfrom_list,dyn_int   pwayfrom_list,   dyn_string          pto_list,dyn_int
pwayto_list,  dyn_int       pid_list,  dyn_int       lkpos_list,dyn_string
link_info,int       reload_connectivity,int       delete_connectivity,string
```

```
ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetDetailedConnectivityBetweenDevices), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSGetDetailedConnectivityBetweenDevices_plsql","(string dfrom, string
dto,dyn_string nfrom_list, dyn_string nto_list, dyn_string
pfrom_list,dyn_int pwayfrom_list, dyn_string pto_list,dyn_int
pwayto_list, dyn_int pid_list, dyn_int lkpos_list,dyn_string
link_info,string ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetDetailedConnectivityBetweenDevices_plsql), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR, "PVSSUpdateMultipleDeviceActive","(string
devicename,int active, int first_rows,int last_rows, string
errMess)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceActive),IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSUpdateMultipleDeviceActiveByType","(string devicename,int active,int
devtype_given,int last_rows, string errMess)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceActiveByType),IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR, "PVSSGetStatusSubsystem","(dyn_string
subsytemname,dyn_int SubsystemStatus)",true,reinterpret_cast<void
(*)()>(PVSSGetStatusSubsystem), IntegerVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetBoardCpntRow_cpntname","(string
cpntname)",true,reinterpret_cast<void (*)()>(PVSSGetBoardCpntRow_cpntname),
TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetHWCpntRow_serialnb","(string
serialnb)",true,reinterpret_cast<void (*)()>(PVSSGetHWCpntRow_serialnb),
TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetBoardCpntRow_cpntid","(int
cpntid)",true,reinterpret_cast<void
(*)()>(PVSSGetBoardCpntRow_cpntid),TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetHWCpntRow_snbid","(int
snbtid)",true,reinterpret_cast<void
(*)()>(PVSSGetHWCpntRow_snbid),TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetMicroConnectivityRow_lkid","(int
lkid)",true,reinterpret_cast<void
(*)()>(PVSSGetMicroConnectivityRow_lkid),TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetMicroConnectivityRow_node","(int
nodeID,int ptnb,int way)",true,reinterpret_cast<void
(*)()>(PVSSGetMicroConnectivityRow_node),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetSpareHWCpntPerType","(string
hwtype)",true,reinterpret_cast<void
(*)()>(PVSSGetSpareHWCpntPerType),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetSpareHWPerType","(string
hwtype)",true,reinterpret_cast<void
(*)()>(PVSSGetSpareHWPerType),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetSpareHWCpntPerLocation","(string
location)",true,reinterpret_cast<void
(*)()>(PVSSGetSpareHWCpntPerLocation),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetDeviceNamesPerLocation","(string
location)",true,reinterpret_cast<void
(*)()>(PVSSGetDeviceNamesPerLocation),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetCpntNamesPerBoard","(string
location)",true,reinterpret_cast<void
(*)()>(PVSSGetCpntNamesPerBoard),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetMicroLkToCpntID","(int cpntid_to,int
motherboard,string ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetMicroLkToCpntID),TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetMicroLkFromCpntID","(int cpntid_from,int
motherboard,string ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetMicroLkToCpntID),TextVar(0)),
```

```
FunctionListRecEx(DYN_VAR,                "PVSSGetBoardCpntPerType","(string
cpnttype)",true,reinterpret_cast<void
(*)()>(PVSSGetBoardCpntPerType),TextVar(0)),
FunctionListRecEx(DYN_VAR,           "PVSSInsertMultipleBoardCpnts","(string
cpntname,string   cpnttype,int   replacable,string   motherboardname,string
serialnb,string     hwtype,string     responsible,string     location,string
comments,int       first,       int       last)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleBoardCpnts),TextVar(0)),
FunctionListRecEx(DYN_VAR,         "PVSSInsertMultipleSpareDevices","(string
hwname,string     hwtype,string     serialnb,string     responsible,string
location,string comments,int first, int last)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleSpareDevices),TextVar(0)),
FunctionListRecEx(DYN_VAR,  "PVSSInsertSpareDevice","(string  hwname,string
hwtype,string serialnb,string responsible,string location,string comments,
int                                    last)",true,reinterpret_cast<void
(*)()>(PVSSInsertSpareDevice),TextVar(0)),
FunctionListRecEx(DYN_VAR,         "PVSSInsertTestFunctionalDevice","(string
location,dyn_string       ipaddList,dyn_string       ipnameList,dyn_string
subnetList,dyn_string      portnbList,dyn_string      port_typeList,dyn_int
port_wayList,             int             last)",true,reinterpret_cast<void
(*)()>(PVSSInsertTestFunctionalDevice),TextVar(0)),
FunctionListRecEx(DYN_VAR,  "PVSSUpdateMultiplePorts","(string  dname,string
pt_nb,string  pt_type,int  pway,int  pxibooting,int  speed,  string  phy,int
first_time,int                         last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultiplePorts),TextVar(0)),
FunctionListRecEx(DYN_VAR,         "PVSSUpdateMultiplePortStatuses","(string
dname,string   pt_nb,string   pt_type,int   admin_status,int   first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultiplePortStatuses),TextVar(0)),
FunctionListRecEx(DYN_VAR,      "PVSSUpdateMultipleAttributeMacIPs","(string
ipadd,string       subnet_mask,string       ipname,int       first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleAttributeMacIPs),TextVar(0)),
FunctionListRecEx(DYN_VAR,          "PVSSUpdateMultipleIPAddresses","(string
ipadd,string                   old_ipadd,int                   first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleAttributeMacIPs),TextVar(0)),
FunctionListRecEx(DYN_VAR,   "PVSSUpdateMultipleCompositeLinkTypes","(string
link_name,string             simple_linkList,int             first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleCompositeLinkTypes),TextVar(0)),
FunctionListRecEx(DYN_VAR,        "PVSSUpdateMultipleLinkTypeNames","(string
oldlink_name,string             oldlink_name,int             first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleLinkTypeNames),TextVar(0)),
FunctionListRecEx(DYN_VAR,
"PVSSUpdateMultipleDeviceTypeAttributes","(string             devtype,string
description,int  nbinput,int  nboutput,string  rgbcolor,int  first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceTypeAttributes),TextVar(0)),
FunctionListRecEx(DYN_VAR,          "PVSSUpdateMultipleDeviceTypes","(string
old_devtype,string                devtype,int                first_time,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceTypes),TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSGetMacroConnectivityRow_nodename",
"(string        nodename,        string        ptnb,int        pway,string
ptype)",true,reinterpret_cast<void
(*)()>(PVSSGetMacroConnectivityRow_nodename), TextVar(0)),
FunctionListRecEx(DYN_VAR,     "PVSSGetLkTypeDecomposition_lknb",     "(int
linktype_nb)",true,reinterpret_cast<void
(*)()>(PVSSGetLkTypeDecomposition_lknb), TextVar(0)),
```

```
FunctionListRecEx(TEXT_VAR,     "PVSSInsertMultipleSparePorts",     "(string
serialnb,string pt_nb,string pt_type,int pway,string bia,string macadd,int
first_time,int                          last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleSparePorts), TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSInsertSparePort", "(string serialnb,string
pt_nb,string     pt_type,int     pway,string     bia,string     macadd,int
last_rows)",true,reinterpret_cast<void          (*)()>(PVSSInsertSparePort),
TextVar(0)),
FunctionListRecEx(DYN_VAR,              "PVSSGetSpareHWPerLocation","(string
location)",true,reinterpret_cast<void
(*)()>(PVSSGetSpareHWPerLocation),TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetFunctionalDeviceStatus",     "(string
dname)",true,reinterpret_cast<void    (*)()>(PVSSGetFunctionalDeviceStatus),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetHWDeviceRow_serialnb",      "(string
serialnb)",true,reinterpret_cast<void   (*)()>(PVSSGetHWDeviceRow_serialnb),
TextVar(0)),
FunctionListRecEx(DYN_VAR,    "PVSSGetFunctionalDeviceByStatus",    "(string
system_name,string                device_status)",true,reinterpret_cast<void
(*)()>(PVSSGetFunctionalDeviceByStatus), TextVar(0)),
FunctionListRecEx(TEXT_VAR,         "PVSSGetHWDeviceStatus",         "(string
serialnb)",true,reinterpret_cast<void         (*)()>(PVSSGetHWDeviceStatus),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetHWLastFunctionalDevice",     "(string
serialnb)",true,reinterpret_cast<void
(*)()>(PVSSGetHWLastFunctionalDevice), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetFunctionalDeviceLastHW",     "(string
functional_name)",true,reinterpret_cast<void
(*)()>(PVSSGetFunctionalDeviceLastHW), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSDeleteLinkRow",     "(int     linkid,int
macro_link)",true,reinterpret_cast<void          (*)()>(PVSSDeleteLinkRow),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,         "PVSSSetToTestUseStatus",         "(string
functional_name,string     user_comments,string     status_datechange,string
serialnb_replacement,string                          testdevice_name,string
replace_date)",true,reinterpret_cast<void    (*)()>(PVSSSetToTestUseStatus),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSUpdateHWBoardCpntStatus",      "(string
serialnb,string new_status,string new_location,string user_comments,string
status_datechange,string    functional_cpntname)",true,reinterpret_cast<void
(*)()>(PVSSUpdateHWBoardCpntStatus), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSReplaceFunctionalBoardCpnt",    "(string
cpnt_name,string new_status,string new_location,string user_comments,string
status_datechange,string                          serialnb_replacement,string
replace_date)",true,reinterpret_cast<void
(*)()>(PVSSReplaceFunctionalBoardCpnt), TextVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSReplaceFunctionalDevice",      "(string
functional_name,string         new_status,string         new_location,string
user_comments,string  status_datechange,string  serialnb_replacement,string
replace_date)",true,reinterpret_cast<void
(*)()>(PVSSReplaceFunctionalDevice), TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSUpdateMultipleBidrectionalLinks", "(string
node_from,string    node_to,string    portnb_from,string    portnb_to,string
portype_from,string  portype_to,int  bidirectional_link,int  first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleBidrectionalLinks), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSUpdateMultipleLkInfoLinks",     "(string
node_from,string    node_to,string    portnb_from,string    portnb_to,string
portype_from,string    portype_to,string    link_info,int    first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleLkInfoLinks), TextVar(0)),
```

```
FunctionListRecEx(TEXT_VAR,     "PVSSUpdateMultipleLkUsedLinks",     "(string
node_from,string    node_to,string    portnb_from,string    portnb_to,string
portype_from,string      portype_to,int      lkused,int      first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleLkUsedLinks), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSUpdateMultipleLkTypeLinks",     "(string
node_from,string    node_to,string    portnb_from,string    portnb_to,string
portype_from,string    portype_to,string    link_type,int    first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleLkUsedLinks), TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSUpdateMultipleAttributesDevices", "(string
devname,int     node,int     promiscuous_mode,string     location,     string
function_list,int    first_rows,int    last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleAttributesDevices), TextVar(0)),
FunctionListRecEx(TEXT_VAR,  "PVSSUpdateMultipleDeviceSystemList",  "(string
devname,string               new_systemList,int               first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceSystemList), TextVar(0)),
FunctionListRecEx(TEXT_VAR,   "PVSSUpdateMultipleDevNamesDevices",   "(string
old_devname,string                devname,int                first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDevNamesDevices), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSUpdateMultipleDTypeDevices",    "(string
devname,string                  devtype,int                first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDTypeDevices), TextVar(0)),
FunctionListRecEx(TEXT_VAR,   "PVSSUpdateMultipleDeviceNodeUsed",   "(string
devname,int                   nodeused,int                first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceNodeUsed), TextVar(0)),
FunctionListRecEx(TEXT_VAR,        "PVSSUpdateMultipleDeviceNodeUsedByType",
"(string    devtype,int    nodeused,int    devtype_given,int    first_rows,int
last_rows)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceNodeUsedByType), TextVar(0)),
FunctionListRecEx(DYN_VAR,        "PVSSGetHistoryOfFunctionalBoardCpntName",
"(string             functional_cpntname,string             min_date,string
max_date)",true,reinterpret_cast<void
(*)()>(PVSSGetHistoryOfFunctionalBoardCpntName), TextVar(0)),
FunctionListRecEx(DYN_VAR,      "PVSSGetHistoryOfHWBoardCpnt",      "(string
serial_nb,int                 snbid,string                min_date,string
max_date)",true,reinterpret_cast<void   (*)()>(PVSSGetHistoryOfHWBoardCpnt),
TextVar(0)),
FunctionListRecEx(DYN_VAR,    "PVSSGetHistoryOfFunctionalDevice",    "(string
functional_name,string                                    min_date,string
max_date)",true,reinterpret_cast<void
(*)()>(PVSSGetHistoryOfFunctionalDevice), TextVar(0)),
FunctionListRecEx(DYN_VAR,        "PVSSGetHistoryOfHWDevice",        "(string
serial_nb,string    min_date,string    max_date)",true,reinterpret_cast<void
(*)()>(PVSSGetHistoryOfHWDevice), TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSGetCpntName_cpntid",          "(int
cpntid)",true,reinterpret_cast<void          (*)()>(PVSSGetCpntName_cpntid),
TextVar(0)),
FunctionListRecEx(INTEGER_VAR,       "PVSSGetCpntID_cpntname",       "(string
cpntname,string                          ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetCpntID_cpntname), TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSGetFunctionalBoardCpntNameLastHW",
"(string                        functional_name)",true,reinterpret_cast<void
(*)()>(PVSSGetFunctionalBoardCpntNameLastHW), TextVar(0)),
FunctionListRecEx(TEXT_VAR,   "PVSSGetFunctionalBoardCpntStatus",   "(string
cpntname)",true,reinterpret_cast<void
(*)()>(PVSSGetFunctionalBoardCpntStatus), TextVar(0)),
```

```
FunctionListRecEx(DYN_VAR,    "PVSSGetFunctionalDeviceByStatus",    "(string
systemname,string                 device_status)",true,reinterpret_cast<void
(*)()>(PVSSGetFunctionalDeviceByStatus), TextVar(0)),
FunctionListRecEx(DYN_VAR,      "PVSSGetHWBoardCpntByStatus",      "(string
cpnt_status)",true,reinterpret_cast<void
(*)()>(PVSSGetHWBoardCpntByStatus), TextVar(0)),
FunctionListRecEx(DYN_VAR,       "PVSSGetHWDeviceByStatus",       "(string
systemname,string                 device_status)",true,reinterpret_cast<void
(*)()>(PVSSGetHWDeviceByStatus), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSGetFunctionalDeviceLastHW",    "(string
functional_name)",true,reinterpret_cast<void
(*)()>(PVSSGetFunctionalDeviceLastHW), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSGetFunctionalDeviceStatus",    "(string
dname)",true,reinterpret_cast<void    (*)()>(PVSSGetFunctionalDeviceStatus),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSGetHWBoardCpntStatus",      "(string
serialnb,int                          snbid)",true,reinterpret_cast<void
(*)()>(PVSSGetHWBoardCpntStatus), TextVar(0)),
FunctionListRecEx(TEXT_VAR,       "PVSSGetHWDeviceStatus",       "(string
dname)",true,reinterpret_cast<void           (*)()>(PVSSGetHWDeviceStatus),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,         "PVSSGetHWLastFunctionalBoardCpntName",
"(string       hw_serialnb,int       hw_snbid)",true,reinterpret_cast<void
(*)()>(PVSSGetHWLastFunctionalBoardCpntName), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSGetHWLastFunctionalDevice",    "(string
functional_name)",true,reinterpret_cast<void
(*)()>(PVSSGetHWLastFunctionalDevice), TextVar(0)),
FunctionListRecEx(INTEGER_VAR,
"PVSSGetMicroConnectBetweenBoardCpntAndMotherBoard", "(string cpnt_from,int
mboard_pid,dyn_string       nfrom_list,string       lastlkfrom_list,string
lastlkto_list,dyn_string         nto_list,dyn_int         pid_list,dyn_int
lkpos_list,string                    ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSGetMicroConnectBetweenBoardCpntAndMotherBoard), IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,    "PVSSGetMicroConnectFromPortid",    "(int
mboardportid_from,int      mboardportid_to,string      cpntname,dyn_string
nfrom_list,string     lastlkfrom_list,string     lastlkto_list,dyn_string
nto_list,dyn_int            pid_list,dyn_int            lkpos_list,string
ErrMess)",true,reinterpret_cast<void  (*)()>(PVSSGetMicroConnectFromPortid),
IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,    "PVSSGetMicroConnectToCpntType",    "(int
mboardportid_from,int      cpnttype_given,string      cpnttype,dyn_string
nfrom_list,string     lastlkfrom_list,string     lastlkto_list,dyn_string
nto_list,dyn_int            pid_list,dyn_int            lkpos_list,string
ErrMess)",true,reinterpret_cast<void  (*)()>(PVSSGetMicroConnectToCpntType),
IntegerVar(0)),
FunctionListRecEx(TEXT_VAR,         "PVSSSwapTwoDevices",         "(string
functional_devname1,string                    functional_devname2,string
comments)",true,reinterpret_cast<void          (*)()>(PVSSSwapTwoDevices),
TextVar(0)),
FunctionListRecEx(INTEGER_VAR,     "PVSSGetPortPerSubsystem",     "(string
dfrom,string           system_name,dyn_string           pfrom_list,string
ErrMess)",true,reinterpret_cast<void       (*)()>(PVSSGetPortPerSubsystem),
IntegerVar(0)),
FunctionListRecEx(INTEGER_VAR,      "PVSSLoadRoutingTable",      "(string
switch_name,dyn_string  destinname_list,dyn_string  nextport_list,dyn_string
ipnext_list,dyn_string  ipdestin_list,dyn_string  subnetnext_list,dyn_string
macaddnext_list,string                ErrMess)",true,reinterpret_cast<void
(*)()>(PVSSLoadRoutingTable), IntegerVar(0)),
FunctionListRecEx(TEXT_VAR,           "PVSSDeletePortRow",           "(int
portid)",true,reinterpret_cast<void (*)()>(PVSSDeletePortRow), TextVar(0)),
```

```
FunctionListRecEx(TEXT_VAR,     "PVSSInsertMultipleBootImages",     "(string
devicename,string       boot_image,string       kernel_image_location,string
initrd_image_location,string    physical_location,string    boot_protocol,int
first_rows,            int            last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertMultipleBootImages), TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSDeleteBootImage",          "(stirng
devicename)",true,reinterpret_cast<void        (*)()>(PVSSDeleteBootImage),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSUpdateBootImage",          "(string
devicename,string       boot_image,string       kernel_image_location,string
initrd_image_location,string                        physical_location,string
boot_protocol)",true,reinterpret_cast<void     (*)()>(PVSSUpdateBootImage),
TextVar(0)),
FunctionListRecEx(INTEGER_VAR,        "PVSSGetPortID_portinfo",        "(int
deviceid,string     port_nb,string     port_type,int     port_way,     string
ErrMess)",true,reinterpret_cast<void         (*)()>(PVSSGetPortID_portinfo),
IntegerVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSInsertSimpleDeviceFunction",    "(string
function_name,int                    last_rows)",true,reinterpret_cast<void
(*)()>(PVSSInsertSimpleDeviceFunction), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSUpdateDeviceFunctionName",     "(string
old_function_name,string   new_function_name   )",true,reinterpret_cast<void
(*)()>(PVSSUpdateDeviceFunctionName), TextVar(0)),
FunctionListRecEx(TEXT_VAR,          "PVSSUpdateHWSerialNB",          "(string
old_serialnb,string        new_serialnb        )",true,reinterpret_cast<void
(*)()>(PVSSUpdateHWSerialNB), TextVar(0)),
FunctionListRecEx(DYN_VAR,     "PVSSGetDeviceNamesPerFunction",     "(string
function_name                                  )",true,reinterpret_cast<void
(*)()>(PVSSGetDeviceNamesPerFunction), TextVar(0)),
FunctionListRecEx(TEXT_VAR,      "PVSSInsertMultipleIPAliases",      "(string
ipaddress,  string  ipname,  string  ipalias,  int  first_time,int  last_rows
)",true,reinterpret_cast<void           (*)()>(PVSSInsertMultipleIPAliases),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,           "PVSSUpdateIPalias",           "(string
old_ipalias,string         new_ipalias         )",true,reinterpret_cast<void
(*)()>(PVSSUpdateIPalias), TextVar(0)),
FunctionListRecEx(DYN_VAR,   "PVSSGetIPAliasesPerIPName",   "(string   ipname
)",true,reinterpret_cast<void             (*)()>(PVSSGetIPAliasesPerIPName),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSGetIPAliasRow",     "(string     ipalias
)",true,reinterpret_cast<void (*)()>(PVSSGetIPAliasRow), TextVar(0)),
FunctionListRecEx(TEXT_VAR,     "PVSSDeleteIPAlias",     "(string     ipalias
)",true,reinterpret_cast<void (*)()>(PVSSDeleteIPAlias), TextVar(0)),
FunctionListRecEx(DYN_VAR, "PVSSGetLkIDsPerLkInfo", "(string lkinfo,string
errmess     )",true,reinterpret_cast<void     (*)()>(PVSSGetLkIDsPerLkInfo),
TextVar(0)),
FunctionListRecEx(DYN_VAR,         "PVSSUpdateMultipleDeviceTypeSystemList",
"(string  devtypename,string  new_systemList,  int  first_rows,int  last_rows
)",true,reinterpret_cast<void
(*)()>(PVSSUpdateMultipleDeviceTypeSystemList), TextVar(0)),
FunctionListRecEx(DYN_VAR,    "PVSSGetAvailableFunctions",    "(int    dummy
)",true,reinterpret_cast<void             (*)()>(PVSSGetAvailableFunctions),
TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSLoadDeviceTypeTable", "(string systemname,
dyn_string               devicetype_name,dyn_int               devicetypeid
)",true,reinterpret_cast<void (*)()>(PVSSLoadDeviceTypeTable), TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSDeleteFunctionalDeviceType",    "(string
devicetype                                     )",true,reinterpret_cast<void
(*)()>(PVSSDeleteFunctionalDeviceType), TextVar(0)),
```

```
FunctionListRecEx(TEXT_VAR,  "PVSSDeleteFunctionalDevice",  "(int  deviceid
)",true,reinterpret_cast<void           (*)()>(PVSSDeleteFunctionalDevice),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSDeleteHWDevice",    "(string    serialnb
)",true,reinterpret_cast<void (*)()>(PVSSDeleteHWDevice), TextVar(0)),
FunctionListRecEx(TEXT_VAR,  "PVSSDeleteSystemName",  "(string  system_name
)",true,reinterpret_cast<void (*)()>(PVSSDeleteSystemName), TextVar(0)),
FunctionListRecEx(TEXT_VAR,       "PVSSUpdateSubsystemName",       "(string
old_systemname,string      new_systemname      )",true,reinterpret_cast<void
(*)()>(PVSSUpdateSubsystemName), TextVar(0)),
FunctionListRecEx(TEXT_VAR,         "PVSSInsertSubsystem",         "(string
system_name,string                 parent_name)",true,reinterpret_cast<void
(*)()>(PVSSInsertSubsystem), TextVar(0)),
FunctionListRecEx(DYN_VAR,    "PVSSGetListOfSubsystems",    "(int    dummy
)",true,reinterpret_cast<void (*)()>(PVSSGetListOfSubsystems), TextVar(0)),
FunctionListRecEx(DYN_VAR,    "PVSSGetSpareHWTypeList",    "(int    dummy
)",true,reinterpret_cast<void (*)()>(PVSSGetSpareHWTypeList), TextVar(0)),
FunctionListRecEx(TEXT_VAR, "PVSSGetSparePortRow_snb","(string snb, string
ptnb,int       port_way,string       port_type)",true,reinterpret_cast<void
(*)()>(PVSSGetSparePortRow_snb),TextVar(0)),
FunctionListRecEx(DYN_VAR,                "PVSSGetPortInfoPerSpare","(string
serialnb)",true,reinterpret_cast<void       (*)()>(PVSSGetPortInfoPerSpare),
TextVar(0)),
FunctionListRecEx(TEXT_VAR,    "PVSSGetBootImageRow_devicename",    "(string
devicename)",true,reinterpret_cast<void
(*)()>(PVSSGetBootImageRow_devicename), IntegerVar(0)),
};
```

Appendix I The CICDB class (for the Python CIC_DB_lib)

```
/*************************************************************************
******/
// Author: L.Abadie
// Code compatible with version: v3.2 of the CIC_DB_lib
// CONFDB class.
/*************************************************************************
*******************/

class CONFDB
{
private:
      string _dbname;
      string _login;
      string _passwd;

public:

CONFDB(string dbname,string login,string passwd);
~CONFDB();
int PyDBConnexion()  ;
      //int PyDBConnexion(string dbname,string login,string passwd);
int PyDBDeconnexion()  ;
string  PyGetDeviceTypeRow(string devitype)  ;
string  PyGetDeviceRow_devicename(string deviname)  ;
string  PyGetDeviceRow_deviceid(int devid)  ;
string PyGetPortRow_pid(int pID)  ;
string  PyGetPortRow_devid(int  devid,string  port_nb,int  port_way,string
port_type)  ;
string PyGetMacroConnectivityRow_lkid(int lkid)  ;
string   PyGetMacroConnectivityRow_node(int   nodeid,string   port_nb,int
way_given,string port_type)  ;
string PyGetIPInfoRow(string IPadd)  ;
string PyGetLkTypeRow_lkname(string lname)  ;
string PyGetLkTypeRow_lknb(int lknb)  ;
vector<int> PyGetPortIDPerDevID(int dID)  ;
vector<string> PyGetDeviceNamesPerType(string dtype)  ;
vector<string> PyGetMacIPAddPerDevID(int dID)  ;
vector<int> PyGetLkToDevID(int nto)  ;
vector<int> PyGetLkFromDevID(int nfrom)  ;
vector<int> PyGetDeviceIDsPerType(string dtype)  ;
string  PyInsertPort(string  devname,string  ptnb,string  port_type,int
port_way,string  bia,string  ipname,string  ipadd,string  subnet,string
macadd,vector<int> param_list, string phy,int last) ;
string PyInsertMultiplePorts(string devname,string ptnb,int port_way,string
port_type,string  bia,string  ipname,string  ipadd,string  subnet,string
macadd,vector<int> param_list, string phy,int first,int last)  ;
string    PyInsertMultipleMacroLinks(string    nfrom,string    nto,string
pfrom,string  pto,string  port_typefrom,string  port_typeto,string  lk_type
,int bidirectional,string link_info,int first,int last)  ;
string       PyInsertMultipleCompositeLinkTypes(string       lktype,string
lktype_list_concan,int first,int last)  ;
string PyInsertMultipleSimpleLinkTypes(string lktype,int first,int last)  ;
string   PyInsertMultipleDeviceTypes(string   sysname,string   devtype,int
nb_in,int nb_out,string devdescr,string rgbcolor,int first,int last) ;
string  PyInsertMacroLink(string  nfrom,string  nto,string  pfrom,string
pto,string   port_typefrom,string   port_typeto,string   lk_type   ,int
bidirectional,string link_info,int last) ;
```

```
string            PyInsertCompositeLinkType(string            lktype,string
lktype_list_concan,int last)  ;
string  PyInsertDeviceType(string  sysname,string  devtype,int  nb_in,int
nb_out,string  devdescr,string rgbcolor,int last)  ;
string PyInsertSimpleLinkType(string lktype,int last)  ;
vector<string>   PyGetConnOutBetweenDeviceDeviceType(string   nfrom,string
nto,int nto_dtype,int reload_connectivity,int delete_connectivity)  ;
vector<string>  PyGetDetailedConnectivityBetweenDevices(string  nfrom,string
nto,int reload_connectivity,int delete_connectivity)  ;
int PyCreateTableSchema()  ;
vector<string>  PyGetConnectivityBetweenDevices(string  nfrom,string  nto,int
reload_connectivity,int delete_connectivity)  ;
vector<string>     PyGetDestinationNamePerDevPort(string     devtype,string
port_nb,string port_type)  ;
int PyGetDeviceID_devicename(string  devname)  ;
string  PyGetDeviceName_deviceid(int devID)  ;
vector<string>  PyGetDetailedConnBetweenDeviceDevType(string  nfrom,string
nto,int dtype_given,int reload_connectivity,int delete_connectivity)  ;
int PyDropTableSchema()  ;
vector<string> PyGetFreeDeviceNamesPerType(string dtype)  ;
int  PyCreateDestinationTable(string dname,int roundtrip)  ;
int  PyCreateRoutingTable(string  devname,  int  roundtrip)  ;
vector<string>          PyGetAllPathsPerDevice(string     systemname,string
devicename,int reload_connectivity,int delete_connectivity)  ;
vector<string> PyMatchDeviceIDDeviceName (string systemname)  ;
vector<string> PyLoadConnectivityTable (string systemname)  ;
vector<string> PyLoadDeviceTypeTable (string systemname)  ;
vector<string> PyLoadLinkTypeTable ()  ;
string  PyUpdateMultiplePorts(string  devicename,  string  port_nb,  int
port_way,string  port_type,int  speed,string  phy,  int  pxi_booting,int
first_time1,int last_rows1);
string         PyUpdateMultipleAttributeMacIPs(string          ip_add,string
subnet_mask,string ipname,int first_time1,int last_rows1);
string  PyUpdateMultipleIPAddresses(string  ip_add,string  oldip_add,int
first_time1,int last_rows1);
string      PyUpdateMultipleCompositeLinkTypes(string      link_name,string
simple_lk_list,int first_time1,int last_rows1);
string    PyUpdateMultipleLinkTypeNames(string    old_link_name,    string
link_name,int first_time1,int last_rows1);
string    PyUpdateMultipleDeviceTypeAttributes(string    devicetype,string
description,int     nbrofinput,int     nbrofoutput,string     rgbcolor,int
first_time1,int last_rows1);
string       PyUpdateMultipleDeviceTypes(string       devicetype_old,string
devicetype,int first_time1,int last_rows1);
string     PyUpdateMultipleDevNamesDevices(string     devicename_old,string
devicename,int first_time1,int last_rows1);
string      PyUpdateMultipleBidirectionalLinks(string      node_from,string
node_to,string  port_nbrfrom,string  port_typefrom,string  port_nbrto,string
port_typeto,int bidirectional_link_used,int first_time1, int last_rows1);
string  PyUpdateMultipleLkUsedLinks(string  node_from,string  node_to,string
port_nbrfrom,string  port_nbrto,string  port_typefrom,string  port_typeto,int
lkused,int first_time1, int last_rows1);
string  PyUpdateMultipleLkTypeLinks(string  node_from,string  node_to,string
port_nbrfrom,string          port_nbrto,string          port_typefrom,string
port_typeto,string link_type,int first_time1, int last_rows1);
      string  PyGetBoardCpntRow_cpntname(string cpntname);
      string  PyGetHWCpntRow_serialnb(string serialnb);
      vector<string> PyGetDeviceNamesPerLocation(string location);
      string  PyGetBoardCpntRow_cpntid(int cpntid);
      string   PyGetHWCpntRow_snbid(int snbid);
      string  PyGetMicroConnectivityRow_lkid(int lkid);
```

```
      string    PyGetMicroConnectivityRow_node(int  nodeid,int  port_nb,int
way_given);
      vector<string>  PyGetBoardCpntPerType(string cpnttype);
      vector<string>  PyGetSpareHWPerLocation(string location);
      vector<string>  PyGetSpareHWCpntPerLocation(string location);
      vector<string>  PyGetSpareHWPerType(string hwtype);
      vector<string>  PyGetCpntNamesPerBoard(string motherboardname);
      vector<string>  PyGetSpareHWCpntPerType(string hwtype);
      vector<int>        PyGetMicroLkFromCpntID(int    cpntid_from,    int
motherboard);
vector<int>  PyGetMicroLkToCpntID(int cpntid_from, int motherboard);
string       PyInsertTestFunctionalDevice(string   location,vector<string>
ipaddList,vector<string>                          ipnameList,vector<string>
subnetList,vector<string>     portnbList,vector<string>     port_typeList,
vector<int> port_wayList,int last);
string    PyInsertMultipleSpareDevices(string  hwname,string  hwtype,string
serialnb,string  responsible,string  location,string  comments,int  first,int
last);
string    PyInsertMultipleBoardCpnts(string  cpntname,string  cpnttype,int
replacable,string  motherboardname,string  serialnb,string  hwtype,string
responsible,string location,string comments,int first,int last);
string      PyInsertMultipleSparePorts(string   serialnb,string   ptnb,int
port_way,string port_type,string bia,string macadd, int first,int last);
string   PyInsertSparePort(string serialnb,string ptnb,int port_way,string
port_type,string bia,string macadd, int last);
string        PyInsertSpareDevice(string    hwname,string    hwtype,string
serialnb,string responsible,string location,string comments,int last);
string   PyInsertMultipleMicroLinks(string  nfrom,string  nto,int  pfrom,int
pto,string lk_type ,int bidirectional,int first,int last);
string   PyGetCpntName_cpntid(int cpntID);
int  PyGetCpntID_cpntname(string  cpntname);
string        PyUpdateMultipleDeviceSystemList(string    devicename,string
new_systemList,int first_time1, int last_rows1);
vector<string>    PyGetMicroConnectFromPortid(int    mboardportid_from,int
mboardportid_to,string cpntname);
vector<string>    PyGetMicroConnectToCpntType(int    mboardportid_from,int
cpnttype_given,string cpnttype);
vector<string>       PyGetMicroConnectBetweenBoardCpntAndMotherBoard(string
cpntname_from,int mboard_portid);
string PyDeleteLinkRow(int linkid,int macro_link);
string PyDeletePortRow(int portid);
vector<string> PyGetLkTypeDecomposition_lknb(int lktypeID);
string PyUpdateMultipleDTypeDevices(string devicename,string devicetype,int
first_time1,int last_rows1);
string  PyUpdateMultipleLkInfoLinks(string  node_from,string  node_to,string
port_nbrfrom,string         port_typefrom,string         port_nbrto,string
port_typeto,string link_info,int first_time1, int last_rows1);
string PyGetBootImageRow_devicename(string functionaldeviname);
string PyDeleteBootImage(string devicename);
string         PyInsertMultipleBootImages(string         devicename,string
boot_image,string kernel_image_location,string initrd_image_location,string
physical_location,string  boot_protocol,int first_rows,int last_rows);
string    PyUpdateBootImage(string    devicename,string    boot_image,string
kernel_image_location,string                    initrd_image_location,string
physical_location,string  boot_protocol);
int  PyGetPortID_portinfo(int  deviceid,  string  port_nb,  string  port_type,
int port_way);
string PyDeleteIPAlias(string ipalias);
string PyInsertSimpleDeviceFunction(string function,int last);
string       PyInsertMultipleFunctionalDevices(string       sysname,string
devname,string  devtype,int  dnode,int  promismode,string  serialnb,string
```

```
hwtype,string     responsible,string     location,string     comments,string
function_list,int first,int last)   ;
string   PyUpdateMultipleAttributesDevices(string   devicename,int   node,int
promiscuous_mode,string  location,string  function_list,int  first_time1,int
last_rows1);
string    PyInsertFunctionalDevice(string    sysname,string    devname,string
devtype,int   dnode,int   promismode,string   serialnb,string   hwtype,string
responsible,string location,string comments,string function_list,int last)
;
vector<int> PyGetLkIDsPerLkInfo(string lkinfo);
string  PyGetIPAliasRow(string ipalias) ;
vector<string> PyGetIPAliasesPerIPName(string IPName);
string PyUpdateIPalias(string old_ipalias,string new_ipalias);
string PyUpdateHWSerialNB(string old_serialnb,string new_serialnb);
string      PyUpdateDeviceFunctionName(string      old_function_name,string
new_function_name);
vector<string> PyGetDeviceNamesPerFunction(string function_name);
string   PyInsertMultipleIPAliases(string   ip_address,string   ipname,string
ipalias,int first,int last);
string     PyUpdateMultipleDeviceTypeSystemList(string     devicetype,string
new_systemList,int first_time1, int last_rows1);
vector<string> PyGetAvailableFunctions();
string PyDeleteHWDevice(string serialnb);
string PyDeleteFunctionalDeviceType(string devicetype);
string PyDeleteFunctionalDevice(int deviceid);
string PyDeleteSystemName(string system_name);
vector<string> PyGetListOfSubsystems();
string PyInsertSubsystem(string system_name,string parent_name);
string PyUpdateSubsystemName(string old_sysname,string new_sysname);
string          PySwapTwoDevices(string          functional_devname1,string
functional_devname2,string comments);
string PyGetFunctionalDeviceStatus(string dname);
string PyGetFunctionalBoardCpntStatus(string cpntname);
string PyGetHWLastFunctionalDevice(string serialnb);
string PyGetHWLastFunctionalBoardCpntName(string hw_serialnb,int hw_snbid);
string PyGetFunctionalDeviceLastHW(string functional_name);
string PyGetFunctionalBoardCpntNameLastHW(string functional_cpntname);
string PyGetHWDeviceStatus(string dname);
string PyGetHWBoardCpntStatus(string serialnb, int snbid);
vector<string>        PyGetHWDeviceByStatus(string        systemname,string
device_status);
vector<string> PyGetHWBoardCpntByStatus(string cpnt_status);
vector<string>        PyGetHistoryOfHWDevice(string        serial_nb,string
min_date,string max_date);
vector<string>        PyGetHistoryOfHWBoardCpnt(string        serial_nb,int
hw_snbid,string min_date,string max_date);
vector<string> PyGetHistoryOfFunctionalDevice(string functional_name,string
min_date,string max_date);
vector<string>                  PyGetHistoryOfFunctionalBoardCpntName(string
functional_cpntname,string min_date,string max_date);
vector<string>    PyGetFunctionalDeviceByStatus(string    systemname,string
device_status);
vector<string> PyGetSpareHWTypeList();
string  PyGetHWDeviceRow_serialnb(string serialnb);
vector<string> PyGetPortInfoPerSpare(string snb) ;
string     PyGetSparePortRow_snb(string     serialnb,string     port_nb,int
port_way,string port_type)  ;

};
```

Appendix J Example of dhcp config file

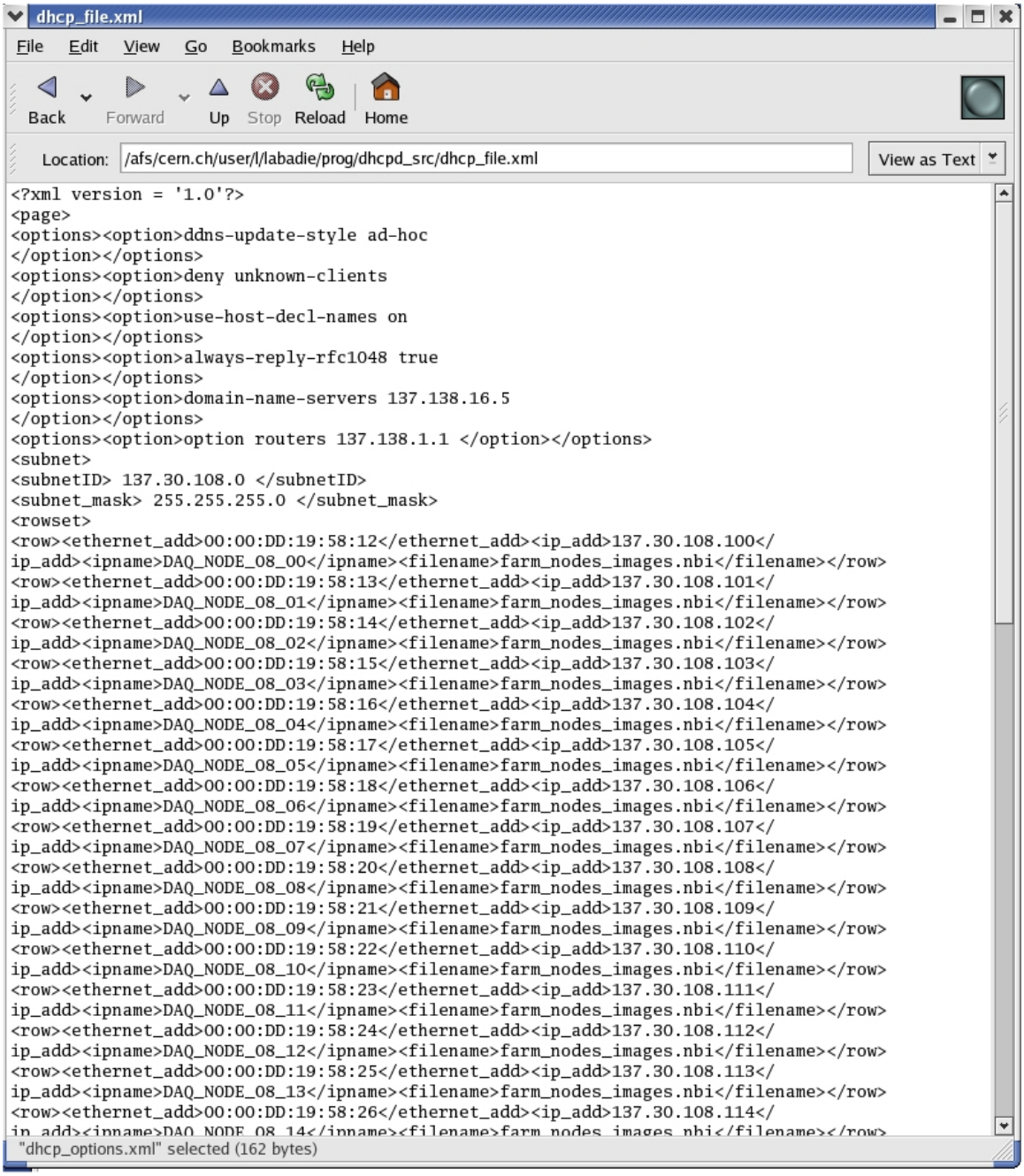


Figure 128. Extract of the dhcp_file.xml.

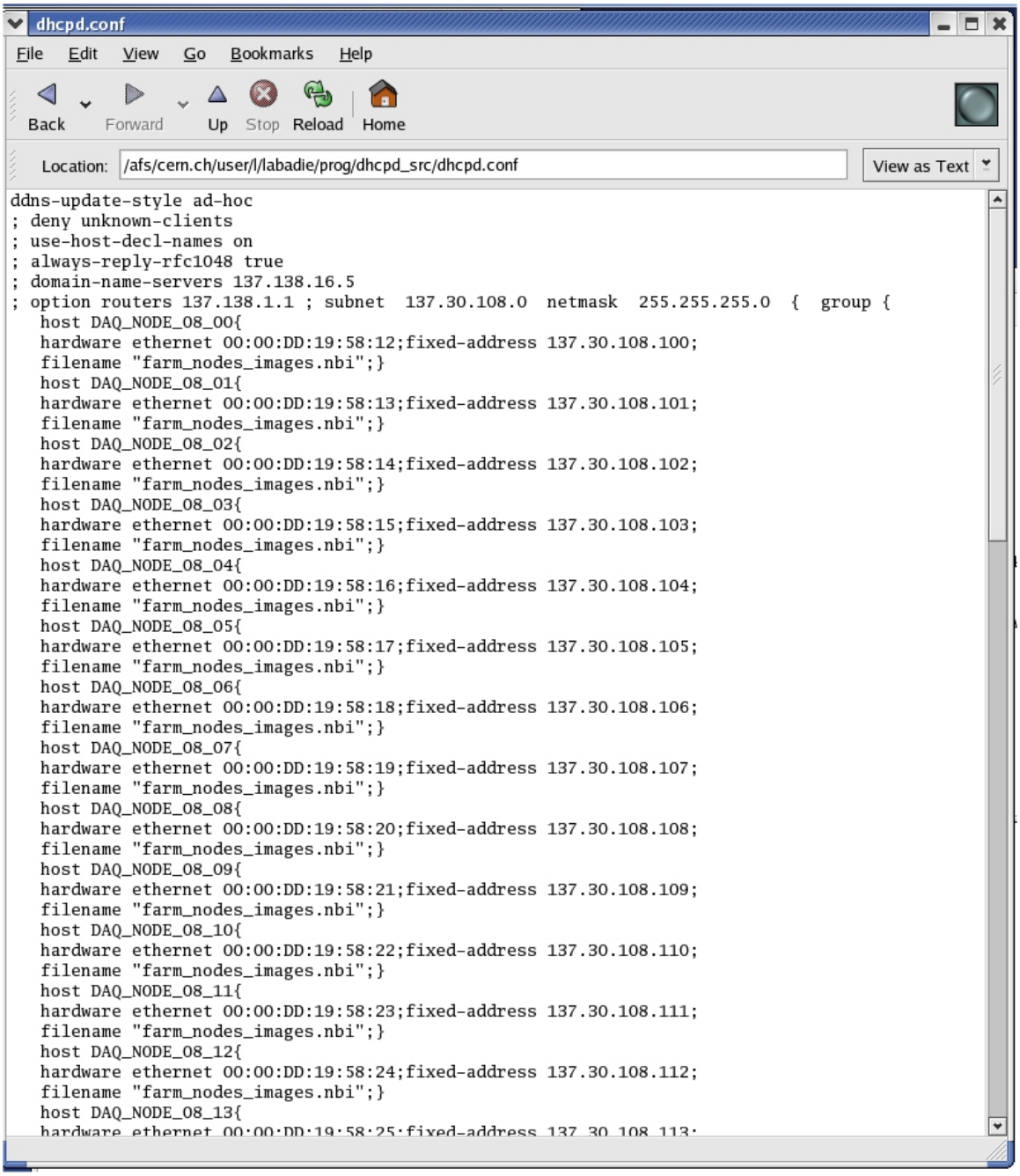


Figure 129. Extract of the dhcp config file.

Appendix J Check that the number of paths for the Flow devices is correct

Let us note N the total number of Flow devices and M the number of connected outputs of a Flow device to the SFCs. We assume that M is the same for all the Flow devices. In our case, M is equal 17. In this context, a valid path is a sequence of distinct nodes where the first node is a Flow device (intermediate node) and the last node is a device of SFC type (it is a hsot node).
Let us show that the number of possible valid paths between a given Flow device and the devices of type SFC, is given by the following formula, N>0 and k corresponds to the number of Flow devices in the path besides the first node:

$$M \sum_{k=0}^{k=N-1} \frac{(N-1)!}{(N-1-k)!}$$

Lemma
If there are N flow devices, the maximum path length is N. It corresponds to the one which goes through all the N Flow devices. Vice-versa, if the maximum path length is N, then there are N Flow devices (indeed if a path length is equal to N, it means that there are N +1 devices in the path). The nodes (except the last ones) in valid paths are Flow devices according to the Flower topology. As the last node in a valid path is a device of type SFC, it means that there are N Flow devices.
So having N Flow devices is equivalent to having the maximum path length equals to N.
Let us count the number of paths group by *p* (where p is the path length) and proof that the number of paths which has a length equal to *p* is M*(N-1)!/(N-p)!

We search for all the paths of length *p*, i.e. paths with p+1 nodes. The first node (Flow device) and last node (SFC) are fixed.
So we need to count the number of ways of obtaining an *ordered* [15] subset of (p-1) elements from a set of (N-1) elements. In other words, it is a permuation of p-1 elements among N-1.
So in total we have M*(N-1)!/(N-p)! possible paths, which correspond to the formula.
Then we just sum up the number of paths group by *p*, over *p*. We then have M∑(N-1)!/(N-p)!, sum over p which is between 1 and N.

As *p* corresponds to the path length, so it is between 1 and N. In the first formula, k is the number of Flow devices in the path, we have k=*p*-1. Indeed if the path length p is 1, it means that we did not select a Flow device as the first one is fixed. So k=0. Same remark if *p*=N.

Thus by replacing N with 6 and M with 17, we obtained 5542 paths.
N.B: the same number of paths group by path hop has been found by the routing algorithm. By symmetry of the topology (rotation), the number of possible valid paths starting from a Flow device and ending at a device of type SFC, is the same for any flower.

[15] The order matters as the path (Flow_0, Flow_1,Flow_2, SFC_1) is not the same as the path (Flow_0, Flow_2,Flow_1, SFC_1). There are two distinct paths.

Appendix L Inserting the connectivity of the HCAL system

```
/************************************************************************
******/
// Author: L.Abadie
// Code compatible with version: v3.2 of the confDB.lib
// Insert the device types, the functional devices, the ports and the link
types for the HCAL system.
/************************************************************************
******/

#include "calo_insertion.h"

int InsertCaloElement()
{
int res_query=0;
int res2=0;
int i=0;
int pos1=0;
int pos2=0;
int loc_pos1=0;
int loc_pos2=0;
int actual_len=0;

char functional_name1[100];
char functional_name2[100];
char device_function="none";
char hw_name1[100];
char hw_name2[100];
char location1[100];
char location2[100];
char devtype[100];
char ptnb[10];
int dtype_case=0;
int j=0;
clock_t start,finish;
double time;
char* ErrMess=(char*)malloc(1001*sizeof(char));
if(ErrMess==NULL)
      return -1;
char* s=(char*)malloc(2001*sizeof(char));
FILE* f;
char* p=NULL;
if(s==NULL)
{
      free(ErrMess);
      return -1;
}


//info about the device type
char
hcal_device_type[6][100]={"HCAL_CHANNEL","HCAL_PMT","HCAL_DAC","HCAL_INTEGR
ATOR","HCAL_FE_CRATE","HCAL_LED"};
int hcal_input_nb[6]={1,1,216,1400,2000,1000};
int hcal_output_nb[6]={1,3,20,20,20,1000};
char
hcal_rgbcolor_type[6][100]={"115,115,114","22,22,22","45,87,96","100,100,10
0","200,200,200","150,150,150"};
```

```
char hcal_description_type[6][100]={"also called cell","pmt for hcal","dac
board for the hcal","integrators for the hcal","fe crates for hcal","led
for hcal"};
start=clock();
// first we insert the device type
for(i=0;i<6;i++)
{
      std::cout<<" devtype="<<hcal_device_type[i]<<" and"<<std::endl;
      std::cout<<" rgbcolor="<<hcal_rgbcolor_type[i]<<" and"<<std::endl;
      std::cout<<"      hcal_description_type="<<hcal_description_type[i]<<"
and"<<std::endl;

      if (i==0)

      res_query=InsertMultipleDeviceTypes("HCAL",hcal_device_type[i],hcal_i
nput_nb[i],hcal_output_nb[i],
hcal_description_type[i],hcal_rgbcolor_type[i],1, 0, ErrMess);
      else
      {
            if(i==5)

      res_query=InsertMultipleDeviceTypes("HCAL",hcal_device_type[i],hcal_i
nput_nb[i],hcal_output_nb[i],
hcal_description_type[i],hcal_rgbcolor_type[i],0, 1, ErrMess);
            else

      res_query=InsertMultipleDeviceTypes("HCAL",hcal_device_type[i],hcal_i
nput_nb[i],hcal_output_nb[i],
hcal_description_type[i],hcal_rgbcolor_type[i],0, 0, ErrMess);

      }
}
i=0;
//then we insert the devices CHANNEL + PMT
//std::cout<<" Device type inserted and errmess="<<ErrMess<<std::endl;

if ((f=fopen( "hcal_conn_1.txt","r"))== NULL)
{
      //printf("Can't open %sn","hcal_conn_1.txt");
      std::cout<<" can't open the file"<<std::endl;
      res_query=DBDeconnexion(ErrMess);
      free(ErrMess);
      free(s);
      Sleep(1000);
      exit(1);
}
else
{
      while (fgets(s,1000,f)!=NULL)
      {
            pos1=strcspn(s," ");
            s[pos1]='\0';
            strcpy(functional_name1,s);

            pos2=strcspn(s+pos1+1," ");
            s[pos1+1+pos2]='\0';
            sprintf(hw_name1,"%s",s+pos1+1);

            actual_len=pos1+1+pos2;
            pos1=strcspn(s+actual_len+1," ");
```

```
            s[pos1+1+actual_len]='\0';
            strcpy(functional_name2,s+actual_len+1);

            actual_len+=pos1+1;
            pos2=strcspn(s+actual_len+1," ");
            s[actual_len+pos2+1]='\0';
            sprintf(hw_name2,"%s",s+actual_len+1);

            loc_pos1=strcspn(functional_name1,"I");
            if(loc_pos1<strlen(functional_name1))

      sprintf(location1,"HCAL_STATION%s",functional_name1+loc_pos1+1);
            else
            {
                  loc_pos1=strcspn(functional_name1,"O");
                  if(loc_pos1<strlen(functional_name1))

      sprintf(location1,"HCAL_STATION%s",functional_name1+loc_pos1+1);
            }

            loc_pos2=strcspn(functional_name2,"I");
            if(loc_pos2<strlen(functional_name2))

      sprintf(location2,"HCAL_STATION%s",functional_name2+loc_pos2+1);
            else
            {
                  loc_pos2=strcspn(functional_name2,"O");
                  if(loc_pos2<strlen(functional_name2))

      sprintf(location2,"HCAL_STATION%s",functional_name2+loc_pos2+1);
            }

            if(i>1480)
            {

      std::cout<<"functional_name1="<<functional_name1<<std::endl;
                  std::cout<<"hw_name1="<<hw_name1<<std::endl;
                  std::cout<<"location1="<<location1<<std::endl;

      std::cout<<"functional_name2="<<functional_name2<<std::endl;
                  std::cout<<"hw_name2="<<hw_name2<<std::endl;
                  std::cout<<"location2="<<location2<<std::endl;


            }
            if(i==0 || (i%100==1 && i>1))
            {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name1,"HCAL_CH
ANNEL",1,0,hw_name1,"hcal_cell","youri","nothing",location1,device_function
,1,0,ErrMess);
            }
            else
            {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name1,"HCAL_CH
ANNEL",1,0,hw_name1,"hcal_cell","youri","nothing",location1,device_function
,0,0,ErrMess);
            }
```

```
            if(strncmp(functional_name2,"HCAL_PMT_I_14_02",16)==0          ||
(i%100==0 && i>0)) //means last one
            {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name2,"HCAL_PM
T",0,0,hw_name2,"hcal_pmt","youri","nothing",location2,device_function,0,1,
ErrMess);

            }
            else
            {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name2,"HCAL_PM
T",0,0,hw_name2,"hcal_pmt","youri","nothing",location2,device_function,0,0,
ErrMess);

            }
            i++;
      }
      //std::cout<<"value of ErrMess"<<ErrMess<<std::endl;
      res2=fclose(f);

      std::cout<<"  first  part  of  devices  inserted"<<ErrMess<<"  and
i="<<i<<std::endl;
      i=0;
}
      //then we insert the devices CHANNEL + PMT

      if ((f=fopen( "hcal_conn_2.txt","r"))== NULL)
      {
            //printf("Can't open %sn","hcal_conn_1.txt");
            std::cout<<" can't open the second file"<<std::endl;
            res_query=DBDeconnexion(ErrMess);
            free(ErrMess);
            free(s);
            Sleep(1000);
            exit(1);
      }
      else
      {

            while (fgets(s,1000,f)!=NULL)
            {
                  pos1=strcspn(s," ");
                  s[pos1]='\0';
                  strcpy(functional_name1,s);
                  pos2=strcspn(s+pos1+1," ");
                  s[pos1+1+pos2]='\0';
                  sprintf(hw_name1,"%s",s+pos1+1);
                  if((p=strstr(functional_name1,"LED"))!=NULL)  //means  it's
a led
                  {
                        loc_pos1=strcspn(functional_name1,"D");
                        if(loc_pos1<strlen(functional_name1))

      sprintf(location1,"HCAL_LED_POS%s",functional_name1+loc_pos1+4);
                        strcpy(devtype,"HCAL_LED");
                        dtype_case=1;
                  }
                  else
                  {
```

```
                        if((p=strstr(functional_name1,"DAC"))!=NULL) //means
it's a DAC
                        {
                              loc_pos1=strcspn(functional_name1,"D");
                              if(loc_pos1<strlen(functional_name1))

      sprintf(location1,"HCAL_DAC_BD%s",functional_name1+loc_pos1+3);
                              strcpy(devtype,"HCAL_DAC");
                              dtype_case=2;
                        }
                        else
                        {
                              if((p=strstr(functional_name1,"INT"))!=NULL)
//means it's a integrators
                              {
                                    loc_pos1=strcspn(functional_name1,"T");
                                    if(loc_pos1<strlen(functional_name1))

      sprintf(location1,"HCAL_INT_BD%s",functional_name1+loc_pos1+1);
                                    strcpy(devtype,"HCAL_INTEGRATOR");
                                    dtype_case=3;
                              }
                              else
                              {

      if((p=strstr(functional_name1,"FE"))!=NULL) //means it's a front-end
                                    {

      loc_pos1=strcspn(functional_name1,"E");

      if(loc_pos1<strlen(functional_name1))

      sprintf(location1,"HCAL_CRATE%s",functional_name1+loc_pos1+1);
                                    }
                                    strcpy(devtype,"HCAL_FE_CRATE");
                                    dtype_case=4;
                              }
                        }
                  }
                  if(i<00)
                  {
                        std::cout<<"hw_name1="<<hw_name1<<std::endl;

      std::cout<<"functional_name1="<<functional_name1<<std::endl;
                        std::cout<<"location1="<<location1<<std::endl;
                  }
                  if(i>1020)
                  {
                        std::cout<<"hw_name1="<<hw_name1<<std::endl;

      std::cout<<"functional_name1="<<functional_name1<<std::endl;
                        std::cout<<"location1="<<location1<<std::endl;
                  }
                  if(i==0)
                  {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name1,devtype,
0,0,hw_name1,devtype,"youri","nothing",location1,device_function,1,0,ErrMes
s);

                  }
```

```
                  else
                  {
                        if(strncmp(functional_name1,"HCAL_FE_04",10)==0)
                        {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name1,devtype,
0,0,hw_name1,devtype,"youri","nothing",location1,device_function,0,1,ErrMes
s);

                        }
                        else
                        {

      res2=InsertMultipleFunctionalDevices("HCAL",functional_name1,devtype,
0,0,hw_name1,devtype,"youri","nothing",location1,device_function,0,0,ErrMes
s);

                        }
                  }
                  i++;
            }
            //std::cout<<"value of ErrMess"<<ErrMess<<std::endl;
            res2=fclose(f);
      }



std::cout<<" second part of devices inserted"<<ErrMess<<std::endl;
// insert the link type
res2=InsertSimpleLinkType("control_link",1, ErrMess);
std::cout<<"Link type inserted and value of ErrMess"<<ErrMess<<std::endl;



// insert the port per device

res_query=InsertMultipleDeviceTypes("HCAL","HCAL_CTRL_PC",4,4,  "ctrl  with
specs master","12,12,12",1, 1, ErrMess);

res2=InsertMultipleFunctionalDevices("HCAL","PCLBHCC","HCAL_CTRL_PC",1,0,"H
W_PCLBHCC","pc","youri","nothing","crate 1",device_function,1,0,ErrMess);
res2=InsertMultipleFunctionalDevices("HCAL","PCLBHCA","HCAL_CTRL_PC",1,0,"H
W_PCLBHCA","pc","youri","nothing","crate 1",device_function,0,0,ErrMess);
res2=InsertMultipleFunctionalDevices("HCAL","PCLBHCI","HCAL_CTRL_PC",1,0,"H
W_PCLBHCI","pc","youri","nothing","crate 1",device_function,0,1,ErrMess);

if ((f=fopen( "hcal_conn_1.txt","r"))== NULL)
{
      //printf("Can't open %sn","hcal_conn_1.txt");
      std::cout<<" can't open the file"<<std::endl;
      res_query=DBDeconnexion(ErrMess);
      free(ErrMess);
      free(s);
      Sleep(1000);
      exit(1);
}
else
{
      i=0;
      while (fgets(s,1000,f)!=NULL)
      {
```

```
            pos1=strcspn(s," ");
            s[pos1]='\0';
            strcpy(functional_name1,s);

            pos2=strcspn(s+pos1+1," ");
            s[pos1+1+pos2]='\0';
            strcpy(hw_name1,s+pos1+1);

            actual_len=pos1+1+pos2;
            pos1=strcspn(s+actual_len+1," ");


            s[pos1+1+actual_len]='\0';
            strcpy(functional_name2,s+actual_len+1);

            actual_len+=pos1+1;
            pos2=strcspn(s+actual_len+1," ");
            s[actual_len+pos2+1]='\0';
            strcpy(hw_name2,s+actual_len+1);



            if(i==0)
            {

      res_query=InsertMultiplePorts(functional_name1,"0",1,2,1000,0,"optica
l","","","","","","T",1,0,ErrMess); //one for pmt

      res_query=InsertMultiplePorts(functional_name1,"1",1,2,1000,0,"optica
l","","","","","","T",0,0,ErrMess); // one for led1

      res_query=InsertMultiplePorts(functional_name1,"2",1,2,1000,0,"optica
l","","","","","","T",0,0,ErrMess); //one for led2
            }
            else
            {

      res_query=InsertMultiplePorts(functional_name1,"0",1,2,1000,0,"optica
l","","","","","","T",0,0,ErrMess); //one for pmt

      res_query=InsertMultiplePorts(functional_name1,"1",1,2,1000,0,"optica
l","","","","","","T",0,0,ErrMess); // one for led1

      res_query=InsertMultiplePorts(functional_name1,"2",1,2,1000,0,"optica
l","","","","","","T",0,0,ErrMess); //one for led2
            }
            if(strcmp(functional_name2,"HCAL_PMT_I_14_02")==0) //means last
one
            {

      res_query=InsertMultiplePorts(functional_name2,"0",1,1,1000,0,"optica
l","","","","","","T",0,0,ErrMess); //one for the channel

      res_query=InsertMultiplePorts(functional_name2,"0",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); // one for the dac

      res_query=InsertMultiplePorts(functional_name2,"1",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); //one for integrators
```

```
      res_query=InsertMultiplePorts(functional_name2,"2",1,2,1000,0,"analog
","","","","","","T",0,1,ErrMess); //one for fe boards

            }
            else
            {

      res_query=InsertMultiplePorts(functional_name2,"0",1,1,1000,0,"optica
l","","","","","","T",0,0,ErrMess); //one for the channel

      res_query=InsertMultiplePorts(functional_name2,"0",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); // one for the dac

      res_query=InsertMultiplePorts(functional_name2,"1",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); //one for integrators

      res_query=InsertMultiplePorts(functional_name2,"2",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); //one for fe boards

            }
            i++;
      }
      std::cout<<"       first       part       of       devices       ports
inserted"<<ErrMess<<std::endl;

      res2=fclose(f);
      i=0;
}

if ((f=fopen( "hcal_conn_2.txt","r"))== NULL)
{
      //printf("Can't open %sn","hcal_conn_1.txt");
      std::cout<<" can't open the second file"<<std::endl;
      res_query=DBDeconnexion(ErrMess);
      free(ErrMess);
      free(s);
      Sleep(1000);
      exit(1);
}
else
{
      i=0;
      while (fgets(s,1000,f)!=NULL)
      {
            pos1=strcspn(s," ");
            s[pos1]='\0';
            strcpy(functional_name1,s);
            pos2=strcspn(s+pos1+1," ");
            s[pos1+1+pos2]='\0';
            strcpy(hw_name1,s+pos1+1);
            if((p=strstr(functional_name1,"LED"))!=NULL) //means it's a led
            {
                  for(j=0;j<100;j++)
                  {
                        sprintf(ptnb,"%d",j);
                        if(i==0 && j==0)

      res_query=InsertMultiplePorts(functional_name1,ptnb,1,1,1000,0,"optic
al","","","","","","T",1,0,ErrMess); //one  to channel
                        else
```

```
      res_query=InsertMultiplePorts(functional_name1,ptnb,1,1,1000,0,"optic
al","","","","","","T",0,0,ErrMess); //one  to channel
                  }

      res_query=InsertMultiplePorts(functional_name1,"0",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); // one for the dac

      res_query=InsertMultiplePorts(functional_name1,"1",1,2,1000,0,"analog
","","","","","","T",0,0,ErrMess); //one for fe boards

            }
            else
            {
                  if((p=strstr(functional_name1,"DAC"))!=NULL)  //means  it's
a DAC
                  {
                        for(j=0;j<216;j++)
                        {
                              if(j<10)
                                    sprintf(ptnb,"00%d",j);
                              else
                              {
                                    if(j<100)
                                          sprintf(ptnb,"0%d",j);
                                    else
                                          sprintf(ptnb,"%d",j);
                              }

      res_query=InsertMultiplePorts(functional_name1,ptnb,1,1,1000,0,"analo
g","","","","","","T",0,0,ErrMess); //one  to led/pmt
                        }
                        for(j=0;j<4;j++)
                        {
                              sprintf(ptnb,"%d",j);

      res_query=InsertMultiplePorts(functional_name1,ptnb,1,2,1000,0,"analo
g","","","","","","T",0,0,ErrMess); // one for the control pc
                        }
                  }
                  else
                  {
                        if((p=strstr(functional_name1,"INT"))!=NULL) //means
it's a integrators
                        {
                              for(j=0;j<1400;j++)
                              {
                                    if(j<10)
                                          sprintf(ptnb,"000%d",j);
                                    else
                                    {
                                          if(j<100)
                                                sprintf(ptnb,"00%d",j);
                                          else
                                          {
                                                if(j<1000)
                                                      sprintf(ptnb,"0%d",j);
                                                else
                                                      sprintf(ptnb,"%d",j);
                                          }
```

```
                                    }

      res_query=InsertMultiplePorts(functional_name1,ptnb,1,1,1000,0,"analo
g","","","","","","T",0,0,ErrMess); //one  to led/pmt
                                    }
                                    for(j=0;j<4;j++)
                                    {
                                          sprintf(ptnb,"%d",j);

      res_query=InsertMultiplePorts(functional_name1,ptnb,1,2,1000,0,"analo
g","","","","","","T",0,0,ErrMess); // one for the control pc
                                    }
                              }
                              else
                              {
                                    for(j=0;j<1500;j++)
                                    {
                                          if(j<10)
                                                sprintf(ptnb,"000%d",j);
                                          else
                                          {
                                                if(j<100)
                                                      sprintf(ptnb,"00%d",j);
                                                else
                                                {
                                                      if(j<1000)
                                                            sprintf(ptnb,"0%d",j);
                                                      else
                                                            sprintf(ptnb,"%d",j);
                                                }
                                          }



      res_query=InsertMultiplePorts(functional_name1,ptnb,1,1,1000,0,"analo
g","","","","","","T",0,0,ErrMess); //one  for LED + PMT

                                    }
                                    for(j=0;j<4;j++)
                                    {
                                          sprintf(ptnb,"%d",j);

      res_query=InsertMultiplePorts(functional_name1,ptnb,1,2,1000,0,"analo
g","","","","","","T",0,0,ErrMess); //one  for the control PC

                                    }

                              }
                        }
                  }

                  i++;
      }

      for(j=1;j<5;j++)
      {
            sprintf(ptnb,"%d",j);

      res_query=InsertMultiplePorts("PCLBHCI",ptnb,1,1,1000,0,"specs","","",
"","","","T",0,0,ErrMess); //one  to led/pmt
            if(j<3)
```

```
            {

      res_query=InsertMultiplePorts("PCLBHCC",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,0,ErrMess); //one  to led/pmt

      res_query=InsertMultiplePorts("PCLBHCA",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,0,ErrMess); //one  to led/pmt
            }
      }
      for(j=3;j<5;j++)
      {

            sprintf(ptnb,"%d_1",j);

      res_query=InsertMultiplePorts("PCLBHCC",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,0,ErrMess); //one  to led/pmt

      res_query=InsertMultiplePorts("PCLBHCA",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,0,ErrMess); //one  to led/pmt
            sprintf(ptnb,"%d_2",j);

      res_query=InsertMultiplePorts("PCLBHCC",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,0,ErrMess); //one  to led/pmt
            if(j==4)

      res_query=InsertMultiplePorts("PCLBHCA",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,1,ErrMess); //one  to led/pmt
            else

      res_query=InsertMultiplePorts("PCLBHCA",ptnb,1,1,1000,0,"specs","","",
,"","","","T",0,0,ErrMess); //one  to led/pmt

      }
      std::cout<<"       second       part       of       devices       ports
inserted"<<ErrMess<<std::endl;

            res2=fclose(f);
}

finish = clock();
time = (double(finish)-double(start))/CLOCKS_PER_SEC;
std::cout<<"time= "<<time <<std::endl;
free(ErrMess);
free(s);
return 0;
}

/*************************************************************************
******/
// Author: L.Abadie
// Code compatible with version:v3.2 of the confDB.lib
// Insert the macro links for the HCAL.
/*************************************************************************
******/


#include "calo_insertion.h"

int  GetIndexForTabCHAR_second(char**   element_List,int  elt_List_nb,  char*
element_searched)
{
```

```
int i=0;
int res=-1;
int stop=1;
int pos2=2;
int pos1=0;
int j=0;
int elt_searched_len=strlen(element_searched);

while(i<elt_List_nb)
{
      if(strncmp(element_List[i],element_searched,elt_searched_len)==0)
      {

            return i;

      }
      i++;
}

return res;
}

int InsertCaloConnectivity()
{
int res_query=0;
int res2=0;
int i=0;
int pos1=0;
int pos2=0;
int loc_pos1=0;
int loc_pos2=0;
int actual_len=0;
char login[30];
char pwd[30];
char dbname[30];
char functional_name1[100];
char functional_name2[100];
char functional_name3[100];
char functional_name4[100];
char functional_name5[100];
char functional_name6[100];
char functional_name7[100];
char hw_name1[100];
char hw_name2[100];
char location1[100];
char location2[100];
char devtype[100];
char ptnb[10];
char dac_ptnb[10];
char fe_ptnb[10];
char int_ptnb[10];
char led1_pnb[10];
char led2_pnb[10];
int led_pnb_bis[104];
char** led_name;
int dtype_case=0;
int led1_idx=0;
int led2_idx=0;
int j=0;
clock_t start,finish;
double time;
```

```
led_name=(char**)malloc(104*sizeof(char*));
if(led_name==NULL)
	return -1;
for(i=0;i<104;i++)
{
	led_name[i]=(char*)malloc(61*sizeof(char));
	if(led_name[i]==NULL)
	{
		for(j=0;j<i;j++)
			free(led_name[j]);
		free(led_name);
		return -1;
	}
}
char* ErrMess=(char*)malloc(1001*sizeof(char));
if(ErrMess==NULL)
{
	for(j=0;j<104;j++)
		free(led_name[j]);
	free(led_name);
	return -1;
}
char* s=(char*)malloc(2001*sizeof(char));
FILE* f;
char* p=NULL;
if(s==NULL)
{
	free(ErrMess);
	for(j=0;j<104;j++)
		free(led_name[j]);
	free(led_name);
	return -1;
}
i=0;
j=3;
std::cout<<" before filling things up"<<std::endl;
while(i<104 && j<29)
{
	if(j<10)
	{
		sprintf(led_name[i],"HCAL_LED1_C_0%d",j);
		i++;
		sprintf(led_name[i],"HCAL_LED1_A_0%d",j);
			i++;
		sprintf(led_name[i],"HCAL_LED2_C_0%d",j);
			i++;
		sprintf(led_name[i],"HCAL_LED2_A_0%d",j);
	}
	else
	{
		sprintf(led_name[i],"HCAL_LED1_C_%d",j);
			i++;
		sprintf(led_name[i],"HCAL_LED1_A_%d",j);
			i++;
		sprintf(led_name[i],"HCAL_LED2_C_%d",j);
			i++;
		sprintf(led_name[i],"HCAL_LED2_A_%d",j);
	}
	led_pnb_bis[i]=0;
```

```
	j++;
	i++;

}


i=0;

if ((f=fopen( "hcal_conn1_v3.txt","r"))== NULL)
{
	//printf("Can't open %sn","hcal_conn_1.txt");
	std::cout<<" can't open the file"<<std::endl;
	res_query=DBDeconnexion(ErrMess);
	free(ErrMess);
	free(s);
	for(j=0;j<104;j++)
		free(led_name[j]);
	free(led_name);
	Sleep(5000);
	exit(1);
}
else
{

	std::cout<<" file opened"<<std::endl;
	i=0;
	while (fgets(s,1000,f)!=NULL)
	{

		pos1=strcspn(s," ");
		s[pos1]='\0';
		strcpy(functional_name1,s);//channel

		pos2=strcspn(s+pos1+1," ");
		s[pos1+1+pos2]='\0';
		sprintf(hw_name1,"%s",s+pos1+1);

		actual_len=pos1+1+pos2;
		pos1=strcspn(s+actual_len+1," ");


		s[pos1+1+actual_len]='\0';
		strcpy(functional_name2,s+actual_len+1);//pmt

		actual_len+=pos1+1;
		pos2=strcspn(s+actual_len+1," ");
		s[actual_len+pos2+1]='\0';
		sprintf(hw_name2,"%s",s+actual_len+1);

		actual_len+=pos2+1;
		pos1=strcspn(s+actual_len+1," ");
		s[actual_len+pos1+1]='\0';
		sprintf(functional_name3,"%s",s+actual_len+1);//dac

		actual_len+=pos1+1;
		pos2=strcspn(s+actual_len+1," ");
		s[actual_len+pos2+1]='\0';
		sprintf(dac_ptnb,"%s",s+actual_len+1);

		actual_len+=pos2+1;
```

```
            pos1=strcspn(s+actual_len+1," ");
            s[actual_len+pos1+1]='\0';
            sprintf(functional_name4,"%s",s+actual_len+1);//integrat

            actual_len+=pos1+1;
            pos2=strcspn(s+actual_len+1," ");
            s[actual_len+pos2+1]='\0';
            sprintf(int_ptnb,"%s",s+actual_len+1);

            actual_len+=pos2+1;
            pos1=strcspn(s+actual_len+1," ");
            s[actual_len+pos1+1]='\0';
            sprintf(functional_name5,"%s",s+actual_len+1);//FE

            actual_len+=pos1+1;
            pos2=strcspn(s+actual_len+1," ");
            s[actual_len+pos2+1]='\0';
            sprintf(fe_ptnb,"%s",s+actual_len+1);

            actual_len+=pos2+1;
            pos1=strcspn(s+actual_len+1," ");
            s[actual_len+pos1+1]='\0';
            sprintf(functional_name6,"%s",s+actual_len+1); //LED1

            actual_len+=pos1+1;
            pos2=strcspn(s+actual_len+1," ");
            s[actual_len+pos2+1]='\0';
            sprintf(functional_name7,"%s",s+actual_len+1); //LED2
            //connectivity between channel and pmt
            if(i==0)
            {
                  std::cout<<"initialization"<<std::endl;

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name2,
"0","0","optical","optical","control_link","none",0,1, 0,ErrMess);
            }
            else

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name2,
"0","0","optical","optical","control_link","none",0,0, 0,ErrMess);

            //connectivity between channel and led (1+2)
            //need to solve the input port for the led

      led1_idx=GetIndexForTabCHAR_second(led_name,104,functional_name6);

      led2_idx=GetIndexForTabCHAR_second(led_name,104,functional_name7);
            sprintf(led1_pnb,"%d",led_pnb_bis[led1_idx]);
            sprintf(led2_pnb,"%d",led_pnb_bis[led2_idx]);

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name6,
"1",led1_pnb,"optical","optical","control_link","none",0,0, 0,ErrMess);

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name7,
"2",led2_pnb,"optical","optical","control_link","none",0,0, 0,ErrMess);
            led_pnb_bis[led2_idx]++;
            led_pnb_bis[led1_idx]++;
            //connectivity between pmt and dac

      res_query=InsertMultipleMacroLinks(functional_name2,functional_name3,
"0",dac_ptnb,"analog","analog","control_link","none",0,0, 0,ErrMess);
```

```
            //connectivity between pmt and fe

      res_query=InsertMultipleMacroLinks(functional_name2,functional_name5,
"1",fe_ptnb,"analog","analog","control_link","none",0,0, 0,ErrMess);

            //connectivity between pmt and int
            if(strcmp("HCAL_CELL_I_14_02",functional_name1)==0)

      res_query=InsertMultipleMacroLinks(functional_name2,functional_name4,
"2",int_ptnb,"analog","analog","control_link","none",0,0, 1,ErrMess);
            else

      res_query=InsertMultipleMacroLinks(functional_name2,functional_name4,
"2",int_ptnb,"analog","analog","control_link","none",0,0, 0,ErrMess);


            i++;
      }
      //std::cout<<"value of ErrMess"<<ErrMess<<std::endl;
      res2=fclose(f);

      std::cout<<" first part of connectivity inserted"<<ErrMess<<" and
i="<<i<<std::endl;
      i=0;
}


      //then we insert the devices CHANNEL + PMT

if ((f=fopen( "hcal_conn2_v3.txt","r"))== NULL)
{
      //printf("Can't open %sn","hcal_conn_1.txt");
      std::cout<<" can't open the file"<<std::endl;
      res_query=DBDeconnexion(ErrMess);
      free(ErrMess);
      free(s);
      for(j=0;j<104;j++)
            free(led_name[j]);
      free(led_name);
      Sleep(5000);
      exit(1);
}
else
{
      i=0;
      actual_len=0;
      std::cout<<" second file opened"<<std::endl;

      while (fgets(s,1000,f)!=NULL && i==-1)
      {

            pos1=strcspn(s," ");
            s[pos1]='\0';
            strcpy(functional_name1,s);//LED

            pos2=strcspn(s+pos1+1," ");
            s[pos1+1+pos2]='\0';
            sprintf(hw_name1,"%s",s+pos1+1);//led hw

            actual_len=pos1+1+pos2;
```

```
            pos1=strcspn(s+actual_len+1," ");
            s[pos1+1+actual_len]='\0';
            strcpy(functional_name2,s+actual_len+1);// dac


            actual_len+=pos1+1;
            pos2=strcspn(s+actual_len+1," ");
            s[actual_len+pos2+1]='\0';
            sprintf(dac_ptnb,"%s",s+actual_len+1);

            actual_len+=pos2+1;
            pos1=strcspn(s+actual_len+1," ");
            s[actual_len+pos1+1]='\0';
            sprintf(functional_name4,"%s",s+actual_len+1);//FE

            actual_len+=pos1+1;
            //pos2=strcspn(s+actual_len+1," ");
            s[actual_len+5]='\0';
            sprintf(fe_ptnb,"%s",s+actual_len+1);

            if(i>1000)
            {


      std::cout<<"functional_name1="<<functional_name1<<std::endl;
                  std::cout<<"dac_ptnb="<<dac_ptnb<<std::endl;

      std::cout<<"functional_name2="<<functional_name2<<std::endl;
                  std::cout<<"fe_ptnb="<<fe_ptnb<<std::endl;

      std::cout<<"functional_name4="<<functional_name4<<std::endl;
            }
            //connectivity between led and dac
            if(i==0)
            {
                  std::cout<<"initialization"<<std::endl;

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name2,
"0",dac_ptnb,"analog","analog","control_link","none",0,1, 0,ErrMess);
            }
            else

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name2,
"0",dac_ptnb,"analog","analog","control_link","none",0,0, 0,ErrMess);
            //connectivity between led and fe
            if(i==0)

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name4,
"1",fe_ptnb,"analog","analog","control_link","none",0,1, 0,ErrMess);
            else
            {
                  if(strcmp(functional_name1,"HCAL_LED2_A_28")==0)

      res_query=InsertMultipleMacroLinks(functional_name1,functional_name4,
"1",fe_ptnb,"analog","analog","control_link","none",0,0, 1,ErrMess);
                  else
      res_query=InsertMultipleMacroLinks(functional_name1,functional_name4,
"1",fe_ptnb,"analog","analog","control_link","none",0,0, 0,ErrMess);

            }
            i++;
```

```
      }
      std::cout<<"second part done"<<ErrMess<<std::endl;
      //between fe and control pc
      res_query=InsertMultipleMacroLinks("HCAL_FE_01","PCLBHCA","0","1","an
alog","specs","control_link","none",1,1, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_FE_02","PCLBHCA","0","2","an
alog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_FE_03","PCLBHCC","0","1","an
alog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_FE_04","PCLBHCC","0","2","an
alog","specs","control_link","none",1,0, 0,ErrMess);
      //between dac and control pc
      res_query=InsertMultipleMacroLinks("HCAL_DAC_01","PCLBHCA","0","3_1",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_02","PCLBHCA","0","3_2",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_03","PCLBHCA","0","4_1",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_04","PCLBHCA","0","4_2",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_05","PCLBHCC","0","3_1",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_06","PCLBHCC","0","3_2",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_07","PCLBHCC","0","4_1",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_DAC_08","PCLBHCC","0","4_2",
"analog","specs","control_link","none",1,0, 0,ErrMess);
      //between int and control pc
      res_query=InsertMultipleMacroLinks("HCAL_INT_01","PCLBHCI","0","1","a
nalog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_INT_02","PCLBHCI","0","2","a
nalog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_INT_03","PCLBHCI","0","3","a
nalog","specs","control_link","none",1,0, 0,ErrMess);
      res_query=InsertMultipleMacroLinks("HCAL_INT_04","PCLBHCI","0","4","a
nalog","specs","control_link","none",1,0,1,ErrMess);
      res2=fclose(f);
      std::cout<<" third part of connectivity inserted"<<ErrMess<<" and
i="<<i<<std::endl;
      i=0;
}
finish = clock();
time = (double(finish)-double(start))/CLOCKS_PER_SEC;
std::cout<<"time= "<<time <<std::endl;

free(ErrMess);
free(s);
for(j=0;j<104;j++)
            free(led_name[j]);
free(led_name);
Sleep(50000);
return 0;
}
```

Glossary

ACS :	ALMA Computing Software. Framework based on CORBA for monitoring and configuring the ALMA project.
ALMA :	Atacama Large Millimeter Array project.
API :	Application Program Interface.
ARP :	Address Resolution Protocol.
CALO :	Calorimeter. One of the LHCb subdetector to permit identification of electrons, hadrons and photons.
CASTOR :	CERN Advanced Storage Manager.
CCPC :	Credit-Card PCs. Small (credit-card size) embedded PCs will be used to provide the necessary local intelligence on an electronics board.
CIC DB :	Configuration Inventory Connectivity DB.
CORBA :	Common Object Request Broker Architecture.
CP :	Controls PC. Linux or Windows PC with PVSS installed used to configure and monitor a set of modules.
CRC :	Contrôle de Redondance Cyclique (CheckSum).
CS :	Controls Switch. Switch installed in the controls network.
CVS :	Concurrent Versions System. CVS is a version control system, used to record the history of your source files.
DAC :	Digital Analog Converter.
DAQ :	Data AcQuisition. System part of the LHCb Online system to transfer data to the storage, via the HLT Trigger.
DBA :	DataBase Administrator.
DBMS :	Database Management Systems.
DHCP :	Dynamic Host Configuration Protocol.
DIM :	Distributed Information Management. Communication system for distributed / mixed environments. It provides a network transparent inter-process communication layer.
DIRAC :	**D**istributed **I**nfrastructure with **R**emote **A**gent **C**ontrol. LHCb distributed computing system to facilitate large-scale simulation and user analysis tasks across computing resources of all participating institutes.
DML :	Data Manipulation Language.
DNS :	Domain Name Server.
DP :	Data Point. PVSS concept to represent an instance of a device type.
DPT :	Data Point Type. PVSS concept to represent the structure of a device type class.
DS :	Distribution Switch. DAQ switch connected to the farm nodes.
ECAL :	Electromagnetic CALorimeter. LHCb subdetector built to identify electrons and to provide energy and momentum measurement.
ECS :	Experiment Control System. Subsystem of the Online system, responsible for configuring and operating the LHCb detector.

EFF : Event Filter Farm. DAQ Farm composed to roughly one thousand of diskless PCs which will run the HLT algorithm.

ERM : Entity Relational Model.

FE : Front-End. Electronics modules used by the subdetectors to process the signal.

Fk : Foreign key.

FPGA : Field Programmables Gates Arrays.

FSM : Finite State Machine. Model used to represent the behavior of the LHCb experiment via states and transitions.

GOL : Gigabit Optical Link. Fast (Gigabit) Optical transmitter which is resistant to high radiation.

HCAL : Hadron CALorimeter. LHCb subdetector built to identify hadrons and to provide energy and momentum measurement.

HEP : High Energy Physics. Field of physics which aims at studying tiny particles interactions.

HLT : High Level Trigger. Physics Algorithm to select the most interesting events. It is executed on the EFF PCs.

HPD : Hybrid-Photon Detector. RICH Module based on pixel technology to provide high precision and low-noise detection of the Cherenkov radiation.

HV : High Voltage. HV type A provides AC voltage between 1kV and 50kV and DC between 1.5kV and 75kV. HV type B provides AC voltage above 50kV and DC above 75kV.

I : Index.

IMS : Information Management System.

INT : INTegrators. An electronics device which integrates the signal and is equivalent to a first-order low-pass filter.

IP : Internet Protocol.

IT : Inner Tracker. LHCb subdetector built to provide precise measurements of the momentum of charged particles.

JDBC : Java DataBase Connectivity.

L0 : Level-0. All the electronics modules which detect and generate signals from the particules and process the signal, before the decision of the L0 Trigger.

LAN : Local Area Network.

LCU : Local Control Unit. Control unit used in the VLT project to supervise a group of modules

LED : Light Emitting Diode.

LHC : Large Hadron Collider. HEP experiment built at CERN and composed of 4 detectors (ATLAS, ALICE, CMS and LHCb). It will be the fastest collider pf the world.

LHCb : Large Hadron Collider beauty. One of the 4 experiments at LHC which aims at studying B-mesons.

LV : Low Voltage. LV type A provides AC voltage between 50V and 500V and DC between 120 kV and 750V. LV type B provides AC voltage between 500V and 1kV and DC between 750V and 1.5kV.

MAC : Media Access Control.

MEP : Multi Event Packet. Protocol implemented by LHCb Online group for which embeds several fragments of events in an IP packet.

NIC : Network Interface Card.

NFS : Network File System.

OCCI : Oracle C++ Call Interface.

OCI : Oracle Call Interface.

ODBC : Open DataBase Connectivity.

ODBMS : Object Database Management Systems.

ODMG : Object Data Management Group.

OID : Object IDentifier.

OPC : OLE for Process Control.

OSPF : Open Shortest Path First.

OSI : Open System Interconnected.

OT : Outer Tracker. LHCb subdetector built to provide precise measurements of the momentum of charged particles.

Perl DBI : Perl DataBase Interface.

PHE : Physique des Hautes Energies. Domaine de la physique qui étudie les interactions entre les particules et la désintégration des particules.

Pk : Primary key.

PL/SQL : Procedural Language/ Structured Query Language.

PMT : PhotoMultiplier Tube. Module used by the calorimeters to transform the light from the photons into electronic signals (photoelectrons).

PS : PreShower. LHCb subdetector which is used by the L0 Trigger and which improve the separation between electrons or photons and hadrons.

PUS : Pile-Up System. LHCb subdetector built to identify and reject bunch crossings where many p-p collisions occurred at the same time. It is part of the L0 Trigger.

PVSS : ProzessVisualisierungs- und SteuerungsSystem. SCADA system from ETM, used by the ECS to monitor and configure the LHCb experiment.

RAC : Real Application Cluster. Oracle features to set up a distributed system of databases and which allows availability of the information.

RDBMS : Relational Database Management Systems.

RICH : Ring Imaging Cherenkov. LHCb detector built to allow charged pion/kaon separation.

RIP : Routing Information Protocol.

RM : Relational Model.

SCADA : Supervisory Control And Data Acquisition. A central system used to supervise a site or a process such as chemical, electrical processes which can execute logical processes without the master computer.

SFC : SubFarm Controller. In the old design of the DAQ system (before the 1MHZ readout), PC used to build and send the events to the farm nodes for HLT processing.

SMI : State Management Interface. A tool for developing control systems based on the concept of Finite State Machines (FSM).

SPD : Scintillator Pad Detector. LHCb subdetector designed to distinguish electrons and photons for the LHCb L0 trigger.

SQL : Structured Language Query.

ST : Silicon Tracker. LHCb detector which consists of the IT and the TT.

TGC : Thin Gas Chambers. Used by the MUON trigger system in the forward region of the ATLAS experiment.

TT : Trigger Tracker. LHCb subdetector built to provide precise measurements of the momentum of charged particles.

TTCtx : Timing Trigger and Control transmitter. Fan-outs or passive switch used to split the signal.

TTCoc : Timing Trigger and Control optical coupler. Optical fan-outs or passive switch used to split the signal.

TTCrx : Timing Trigger and Control receiver. Chips located on the boards to receive the clock.

TFC : Timing Fast Control. LHCb system part of the Online which distributes the clock and forwards the decision of the L0 Trigger.

U : Unique.

VELO : VErtex Locator. LHCb subdetector built to determine the position of the collision and the secondary vertices that characterize the events that could display CP violation.

VLT : Very Large Telescope.

XML : Extensible Markup Language.

XSLT : Extensible Stylesheet Language Transformation.