THESE

Présentée
pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE LOUIS PASTEUR DE STRASBOURG

par

Antoni DÍAZ

LOGICIEL DE CONTROLE ET COMMANDE ET
BASE DE DONNEES ORIENTEE OBJET :
APPLICATION DANS LE CADRE DE LA MISE EN ŒUVRE D'UN
ACCELERATEUR DE PARTICULES, LE VIVITRON

---

*Soutenue le **11 janvier 1996** devant la commission d'examen :*

F.A. BECK	*Directeur de Thèse*
M.M. ALEONARD	*Rapporteur externe*
H. SERGOLLE	*Rapporteur externe*
J.L. RIESTER	*Rapporteur interne*
L. MICHEL	

Je remercie tous ceux qui, de près ou de loin, m'ont soutenu et ont eu le courage de me supporter pendant mes moments difficiles.
C'est à eux que je dédie ce manuscrit.

Remerciements

Je remercie vivement Monsieur F.A. BECK, Directeur du Centre de Recherches Nucléaires de Strasbourg, pour avoir accepté de diriger ce travail.

J'exprime toute ma reconnaissance envers Madame M.M. ALEONARD, Monsieur H. SERGOLLE et Monsieur J.L. RIESTER pour avoir accepté de faire partie du jury de thèse.

Je tiens à remercier tout particulièrement Monsieur J.C. MARSAUDON pour ses conseils pertinents et pour son accueil chaleureux.

Je remercie également Monsieur Laurent MICHEL qui par sa patience de tous les instants et ses conseils a su guider mes travaux. Ses idées, ses suggestions et ses travaux ont conduit à la naissance et à la maturation du nouveau système informatique de contrôle et commande.

Je ne peux oublier le promoteur de ce système, Monsieur Bernard HUMBERT qui n'a jamais hésité à me consacrer le plus précieux de son temps pour m'offrir ses généreux conseils. C'est lui qui a pris le risque de m'intégrer au sein de l'équipe de Contrôle et Commande. Je ne peux que le remercier.

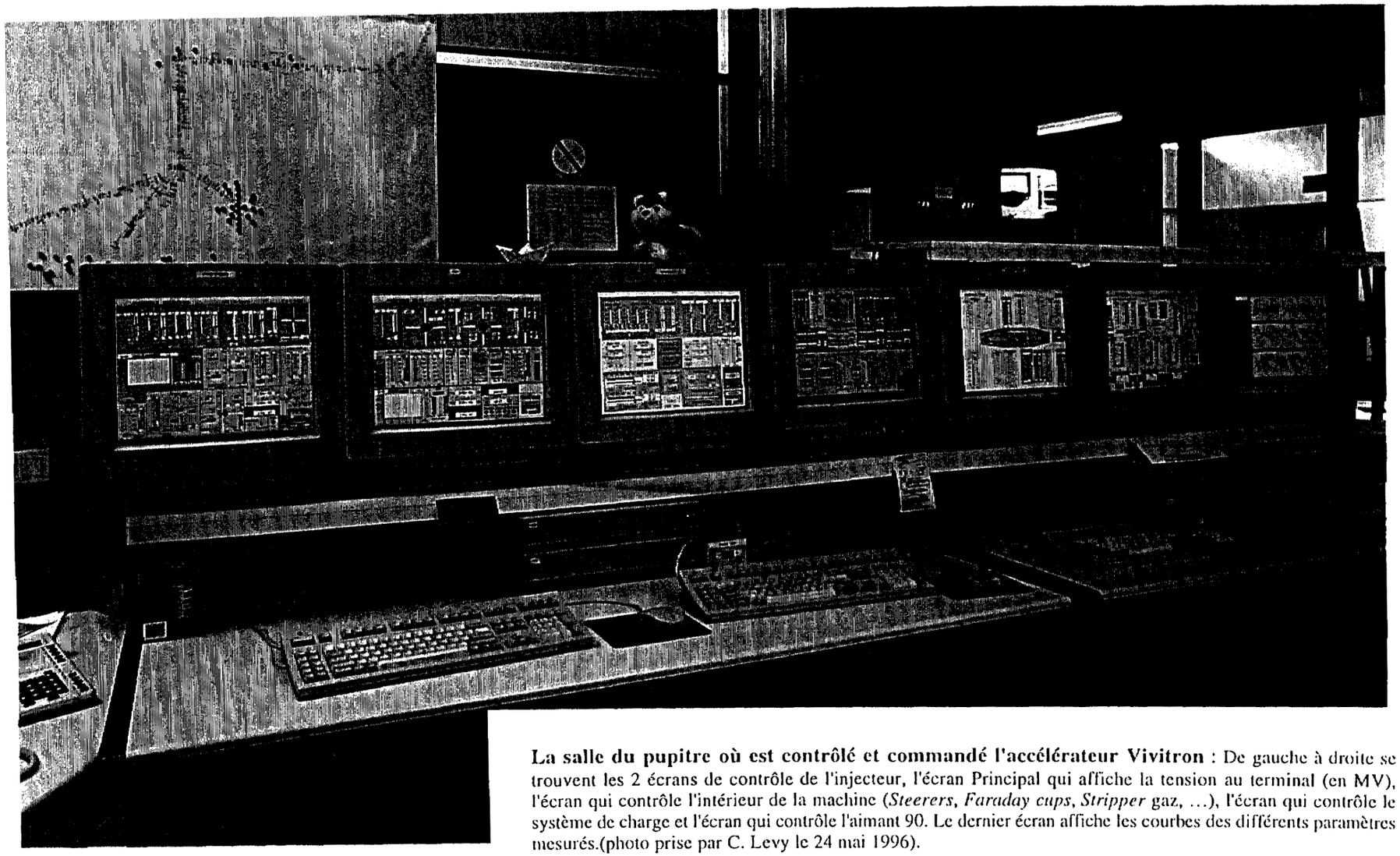
J'exprime aussi toute ma gratitude à cette sympathique équipe de Contrôle et Commande qui avec ses informaticiens et ses électroniciens a su maintenir le cap envers et contre tout pour arriver à faire marcher et à contrôler cette machine, parfois "capricieuse", appelée Vivitron. Ce sont Messieurs R. BAUMANN, A. JBIL, E. KAPPS, R. KNAEBEL, J. PERSIGNY, G. PREVOT, G. SCHWARTZ, Y. STAMM.

Je remercie tous les membres de l'équipe Vivitron. Le système informatique de contrôle et commande a été testé et mis au point avec leur collaboration.

Table des Matières

Liste des Figures

Liste des Tableaux



La salle du pupitre où est contrôlé et commandé l'accélérateur Vivitron : De gauche à droite se trouvent les 2 écrans de contrôle de l'injecteur, l'écran Principal qui affiche la tension au terminal (en MV), l'écran qui contrôle l'intérieur de la machine (*Steerers*, *Faraday cups*, *Stripper* gaz, ...), l'écran qui contrôle le système de charge et l'écran qui contrôle l'aimant 90. Le dernier écran affiche les courbes des différents paramètres mesurés.(photo prise par C. Levy le 24 mai 1996).

Introduction

Le physicien anglais Ernest Rutherford réalise la première réaction nucléaire en 1919. Depuis, la motivation de tout physicien nucléaire est de connaître, de façon toujours plus fine, la structure du noyau de l'atome et les mécanismes de réaction. Les nombreux résultats expérimentaux obtenus dans ce domaine ont contribué à faire de passionnantes découvertes, mais ils ont aussi soulevé de nouvelles questions dont les réponses nécessitent des instruments de plus en plus complexes. Les grands instruments de recherche en physique nucléaire sont les accélérateurs de particules. Les progrès technologiques ont permis aujourd'hui la construction du Vivitron, un accélérateur électrostatique prévu pour atteindre une tension maximale de 35 MV. Le Vivitron a été conçu et réalisé au Centre de Recherches Nucléaires de Strasbourg et sa construction s'appuie sur de nombreuses innovations techniques dans le domaine des accélérateurs électrostatiques.

Jusqu'à ce jour, la conduite des accélérateurs au CRN faisait appel au doigté et au talent des spécialistes qui maniaient les différentes manettes de contrôle de la machine. La notion d'informatique et d'électronique embarquée était inexistante. Ainsi, certaines informations fournies par les données physiques étaient amenées directement au pupitre de commande au moyen d'énormes torons de câbles. La mise en route d'une machine comme le Vivitron nécessite un système de contrôle et commande particulier. Près de 1500 paramètres doivent être contrôlés et commandés sur une surface approximative de 2400 m^2 pour le suivi du faisceau depuis la source jusqu'à la cible du physicien. Plus de la moitié de ces paramètres sont situés à l'intérieur de la cuve (tension variant de 0 à 35 MV) et de l'injecteur (tension variant de 0 à 380 kV) ce qui distingue le Vivitron de la plupart des autres types d'accélérateurs. Ce sont des environnements électromagnétiques extrêmement sévères, surtout l'intérieur de la cuve : vide puis surpression de 6 bars de SF_6, tensions très élevées, tenue aux décharges électriques ($dI/dt > 100$ kA/µs). Quatre endroits sensibles doivent ainsi être contrôlés depuis l'extérieur : l'injecteur, les deux extrémités de l'accélérateur et le centre qui est porté à la tension maximale. Toute l'information fournie par la machine doit être centralisée au pupitre de contrôle situé à l'extérieur du hangar du Vivitron. Le nombre de paramètres à gérer et la complexité d'un tel accélérateur ont mis en évidence la nécessité d'informatiser son système de contrôle et commande.

Le projet Vivitron est proposé en 1981 mais ce n'est qu'en 1985 que la première pierre est posée. Deux ans plus tard, une équipe réduite entame l'étude du système de contrôle et commande. Son activité s'exerce dans deux directions : la mise en œuvre d'un système permettant la conduite des essais en tension du générateur et son évolution vers un système de contrôle définitif pour la phase d'exploitation de l'accélérateur. La phase de test correspond à la période 1991-1992 pendant laquelle l'architecture matérielle et logicielle du système de contrôle et commande va être validée [1]. Le système matériel se décompose en trois niveaux :

- les capteurs et actionneurs près des équipements ;
- les ordinateurs frontaux (dont certains sont embarqués, ceux des extrémités et du centre de l'accélérateur et ceux de l'injecteur) réalisés au standard VME, rassemblant et filtrant les informations et assurant certaines prises de décision ;
- les stations de travail présentant les informations sous forme graphique, signalant les alarmes ou permettant de commander des interventions sur les équipements.

Cette architecture est distribuée sur un réseau Ethernet. Les liaisons entre l'intérieur et l'extérieur de la machine s'opèrent par fibre optique depuis l'injecteur et les extrémités, et par faisceau laser depuis le centre, toute liaison galvanique étant exclue. L'équipe de contrôle et commande actuelle se crée en octobre 1992 pour faire les choix informatiques qui vont inaugurer la phase d'exploitation.

Mon travail de thèse concerne la phase d'exploitation du projet Vivitron. L'objectif de ce manuscrit est double : présenter ma contribution au projet Vivitron et exposer de manière synthétique le système informatique conçu pour cette phase d'exploitation.

La démarche que nous adopterons va consister à présenter, en premier lieu, un bref historique sur les accélérateurs électrostatiques et à exposer le principe de fonctionnement de ce type de machines. Ainsi nous pourrons mieux cerner les nouveaux concepts introduits dans le projet Vivitron et les difficultés liées à leurs mise en application. Nous justifierons aussi les critères qui nous ont poussé à choisir une voie plutôt qu'une autre dans la construction de notre système de contrôle et commande.

En deuxième lieu, nous décrirons l'environnement logiciel et matériel du système de contrôle et commande de l'accélérateur Vivitron. Nous introduirons les communications dans un environnement UNIX en portant une attention particulière aux protocoles Internet et au modèle client-serveur X-Window. Nous présenterons les outils logiciels qui ont été choisis pour construire l'architecture informatique du système de contrôle et commande, à savoir : le système temps réel VxWorks, déjà utilisé pour l'acquisition de données des multidétecteurs (EUROGAM, DIAMANT, DEMON, ICARE), la base de données orientée objet O_2 et l'interface graphique SL-GMS. Du point de vue matériel, nous passerons en revue la spécification VME ainsi que les cartes d'interface du commerce utilisées dans le projet Vivitron.

Enfin, nous aborderons la partie la plus importante de cette thèse : la réalisation du logiciel de contrôle et commande pour la phase d'exploitation de l'accélérateur Vivitron. Nous présenterons un premier logiciel qui a permis le démarrage de l'accélérateur et la première expérience de physique nucléaire en juillet 1994. Ce travail, auquel j'ai apporté ma contribution dans l'aspect communication UNIX/VxWorks, a été réalisé en collaboration avec d'autres informaticiens. L'expérience acquise lors de cette première phase nous a poussé à développer un deuxième ensemble informatique qui doit être le système utilisé pour l'exploitation du Vivitron. L'accent est plus particulièrement porté sur la partie logicielle qui m'a été confiée et sur son intégration dans la base de données O_2, élément central du nouveau système.

— A —

Objectifs

1 Un accélérateur de particules de nouvelle génération : le Vivitron

Les accélérateurs électrostatiques ont toujours été étroitement liés à l'étude et au développement de la physique nucléaire. Ces machines disposent d'une excellente résolution en énergie ($\Delta E/E \approx 10^{-4}$) et l'énergie peut être changée rapidement, ce qui permet, entre autres, de bien explorer les fonctions d'excitation. En disposant de plusieurs sources d'ions à l'injecteur, le faisceau d'ions peut être changé en un temps très restreint. Grâce à la qualité optique du faisceau, l'extension géométrique peut être diminuée ou la divergence angulaire minimisée. Ce sont ces qualités qui font de l'accélérateur électrostatique la machine la mieux adaptée à l'étude de la structure nucléaire.

1.1 Un peu d'histoire

L'histoire des accélérateurs [2] [3] commence en 1919 quand le physicien anglais Ernest Rutherford désintègre l'atome d'azote en utilisant une source artificielle de particules alpha. Mais il avait évalué l'ordre de grandeur des énergies de liaison dans le noyau à plusieurs MeV, ce qui était inaccessible pour les dispositifs expérimentaux de l'époque.

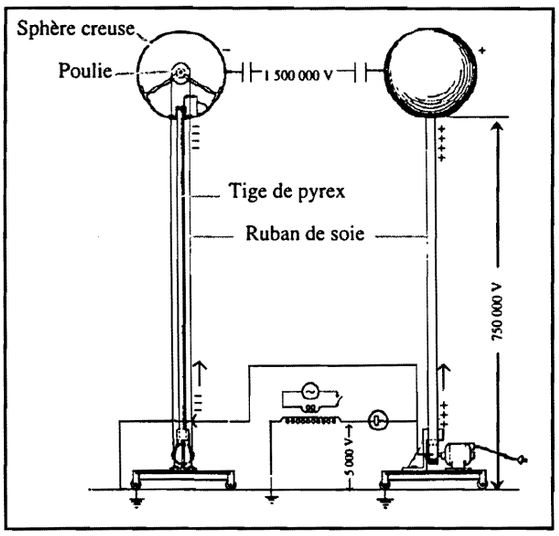


Figure A.1 - Principe du générateur électrostatique Van de Graaff (1,5 MV)

En 1932, à l'université de Cambridge, Cockcroft et Walton, réussirent à désintégrer l'atome de lithium en accélérant des protons à une énergie de seulement 400 keV avec un accélérateur de 600 kV. Presque en même temps, le physicien américain Van de Graaff décrit le premier générateur électrostatique de 1,5 MV (figure A.1). Cette machine se composait de deux sphères creuses en cuivre montées sur deux tiges de pyrex. Chaque sphère était chargée, l'une négativement et l'autre positivement, grâce à une courroie constituée par un ruban de soie tournant autour d'un système de deux poulies. Les charges étaient déposées sur la courroie près de la poulie motrice (placée à la base

de la tige de pyrex) à l'aide d'une alimentation de 10 kV. La récupération des charges électriques s'effectuait par un ensemble de pointes situées à l'intérieur de chaque sphère.

En 1933, R. G. Herb, en collaboration avec D. B. Parkinson et D. W. Kerst, construisit à l'université de Wisconsin le premier accélérateur pressurisé, atteignant une tension de 400 kV. Il utilisait un réservoir contenant comme gaz diélectrique de l'air sec sous pression et permit ainsi de s'affranchir des problèmes d'isolation électrique dus au fonctionnement à l'air libre (humidité). Le tube sous vide permettait aux particules de se déplacer sans collision ni ralentissement ou déviation.

En 1937, la même équipe réalisa un accélérateur horizontal pressurisé de 2,4 MV (figure A.2). Ce fut une machine qui ouvrit une nouvelle voie puisque, pour la première fois, le potentiel électrostatique était réparti uniformément le long d'une colonne grâce à des anneaux métalliques convenablement espacés. La connexion électrique était assurée par le courant corona d'un système pointe-plan. De cette façon, on obtenait une région à champ uniforme dans laquelle se trouvaient le système de charge, la colonne isolante et le tube accélérateur.

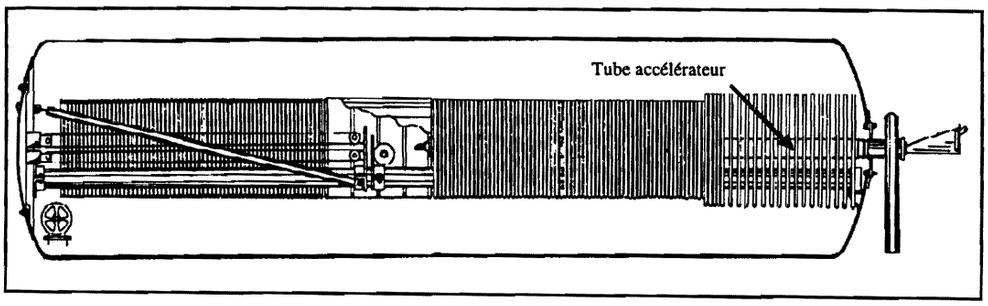


Figure A.2 - L'accélérateur Van de Graaff de Wisconsin à électrodes cylindriques (2,4 MV)

En 1940, R. G. Herb introduisit deux étages d'électrodes cylindriques pour augmenter le potentiel et atteindre 4,5 MV (figure A.3). La même année, Ashby et Hanson inventèrent le système de régulation de la tension du terminal par triode corona, utilisé de nos jours dans tous les accélérateurs électrostatiques, et le système des pointes corona fut remplacé par une chaîne de résistances qui permit l'obtention d'un système indépendant de la pression du gaz isolant.

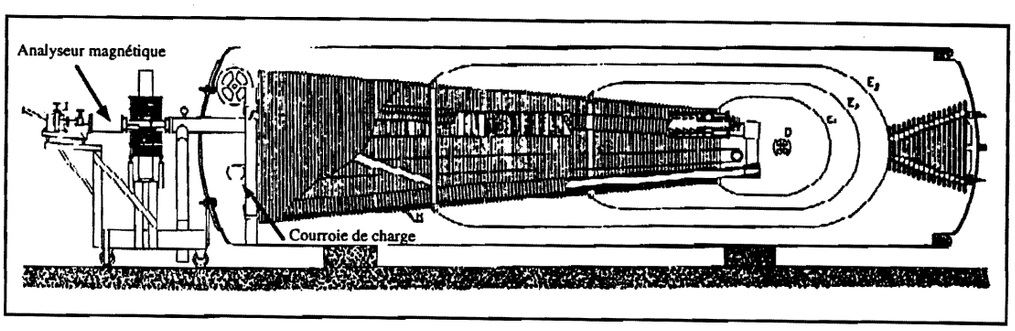


Figure A.3 - L'accélérateur Van de Graaff de Wisconsin (4,5 MV)

En 1947, R. J. Van de Graaff, D. Robinson et J. G. Trump créèrent la société High Voltage Engineering Corporation (HVEC). Un des premiers modèles fut le CN qui atteignait 6,0-6,5 MV. En 1958 Van de Graaff réalisa un nouveau type d'accélérateur : le Tandem Van de Graaff [4], pour le laboratoire canadien de Chalk River. Sa principale nouveauté était un double étage d'accélération

(figure A.4). Cette machine de 5 MV, qui sera commercialisée sous le nom de EN, fournissait des protons de 10 MeV. Le EN fut amélioré pour donner naissance au modèle King (FN) dont la tension nominale atteignait 9 MV.

En 1965, Herb fonda la société NEC (*National Electrostatic Corporation*) qui contribua de façon significative au perfectionnement des accélérateurs électrostatiques afin d'obtenir des tensions extrêmement élevées. HVEC lança ainsi le modèle Empereur (MP) avec une tension nominale de 10 MV. Depuis 1970, la courroie du système de charge a été remplacée dans certaines machines par un système à chaînes (de type Pelletron ou Laddertron). Par la suite, les trois modèles de l'accélérateur Tandem classique EN, FM, MP, ont été utilisés pour réaliser des accélérateurs Tandem à trois étages d'accélération en couplant deux accélérateurs de même modèle ou un injecteur avec un accélérateur. Des machines, qui peuvent être qualifiées de super-Tandems, existent à Jaeri (Japon), 20 MV, à Oak Ridge (États Unis), 25 MV, à Buenos Aires (Argentine), 20 MV, ou à Daresbury (Royaume Uni), 20 MV. Ce dernier a été récemment arrêté. Le nouvel accélérateur Vivitron à Strasbourg assure la relève pour les physiciens anglais.

Depuis ses débuts en 1945, une grande partie de l'histoire de la recherche nucléaire au Centre de Recherches Nucléaires de Strasbourg (CRN) s'est développée autour de ces accélérateurs électrostatiques. En 1959 le CNRS décide de doter Strasbourg d'un accélérateur de type CN (5,5 MV) qui fonctionne encore aujourd'hui pour des essais qui permettent le développement du Vivitron. D'autres machines suivirent et en 1968, il fut décidé d'installer un Tandem du type Empereur (10 MV) que l'ingénieur responsable, M. Letournel, améliora pour aboutir à une tension maximale de 18 MV en 1984. Il y avait déjà, dans ces modifications, l'amorce d'une machine entièrement nouvelle.

La longue expérience acquise dans ce domaine a conduit à la construction d'un nouveau super-Tandem de 35 MV, le Vivitron, présenté pour la première fois par son inventeur, M. Letournel, en 1981. L'énergie maximale atteinte par cette machine devrait être de 20 MeV/nucléon pour les ions légers et environ 5 MeV/nucléon pour les ions les plus lourds. L'intensité prévue est de 10^{12} pps (particules par seconde) pour les ions légers et de 10^{9} pps pour les ions lourds [4]. En juillet 1994, 10 ans après la décision de construire le Vivitron, a eu lieu la première expérience de physique nucléaire.

1.2 Principe de fonctionnement d'un accélérateur de type Tandem

1.2.1 Le système de charge du générateur de tension

Une alimentation haute tension fournit des charges positives à un peigne de charge qui, par frottement, les dépose sur une courroie isolante. Celle-ci est entraînée mécaniquement par deux tambours disposés l'un à la masse et l'autre au terminal haute tension. Elle apporte les charges de façon continue au centre de l'accélérateur où se trouve le terminal haute tension chargé positivement. A ce niveau, un peigne de décharge permet de les récolter. L'entrée et la sortie du réservoir étant à la masse, il s'établit deux différences de potentiel sortie-terminal et entrée-terminal. Ces différences de potentiel sont réparties dans les deux étages de façon continue tout au long du tube accélérateur par l'intermédiaire d'une chaîne de résistances.

1.2.2 L'injecteur

Une source d'ions, placée sur une plate-forme haute tension à l'extérieur du réservoir sous pression, émet un faisceau d'ions positifs. Un dispositif contenant de l'hydrogène attache quelques électrons aux ions positifs pour les transformer en ions négatifs. Le faisceau d'ions négatifs ainsi créé est analysé en masse par un dipôle magnétique (angle de 35°) et injecté à l'entrée de la cuve qui se trouve au potentiel de la terre (figure A.4).

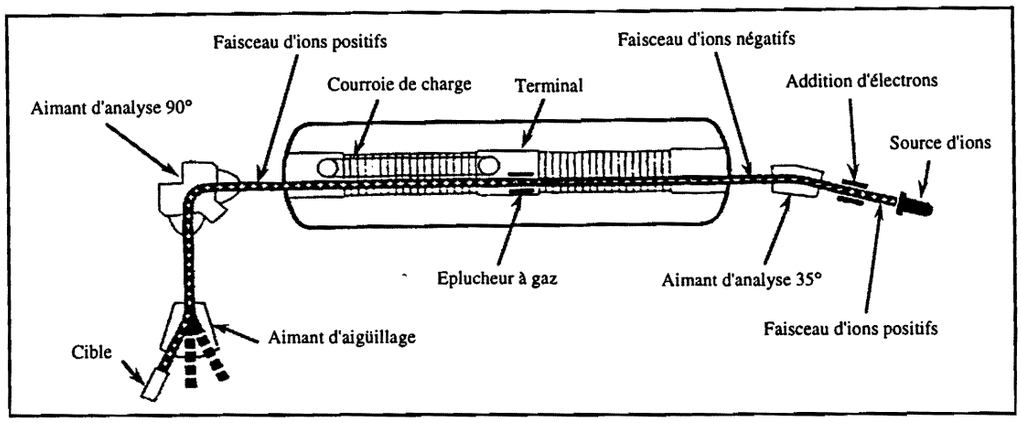


Figure A.4 - Le Tandem : accélérateur à double étage d'accélération

1.2.3 Le faisceau

Dans le cas d'un accélérateur électrostatique classique, des ions positifs sont produits dans le terminal haute tension et accélérés, sous l'effet du champ électrique, vers le potentiel de la terre. Dans le cas d'un Tandem (figure A.4), les ions négatifs subissent une première accélération dans un tube sous vide vers l'électrode terminale portée à une haute tension $+U$. Au cours de leur passage dans le terminal, ces ions traversent un éplucheur (à gaz ou à feuilles de carbone) qui leur arrache plusieurs électrons. Les ions positifs multichargés ainsi produits sont alors repoussés par cette même électrode dans un tube identique au précédent. De la sorte, une énergie de $(1 + q)\cdot U$ est obtenue, où q est la charge des ions positifs.

1.3 Le Vivitron

Cet accélérateur est principalement constitué d'un réservoir de forme biconique de 51 mètres de long et de 8,44 mètres de diamètre au centre [5] qui contient comme gaz diélectrique de l'hexafluorure de soufre (SF_6) avec une surpression nominale d'environ 7 bars actuellement. Toutes les caractéristiques électriques sont très classiques, en particulier pour le tube accélérateur qui est de type standard, mais le principe général et quelques règles de construction sont nouveaux. Dans la suite de l'exposé nous allons utiliser les abréviations définies ci-après :

- BE, pour Basse Energie ;
- HE, pour Haute Energie ;
- SM*i*, pour Section Morte numéro *i*.

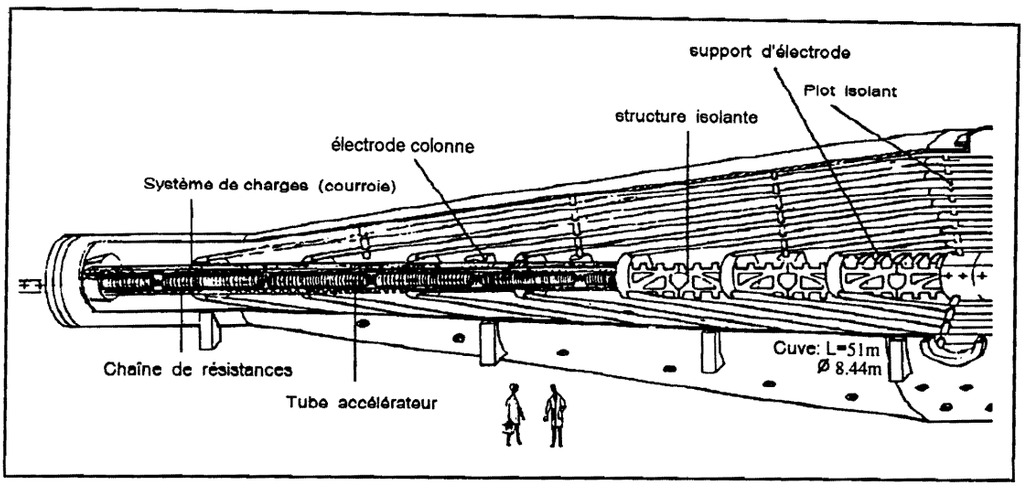


Figure A.6 - La structure du Vivitron

1.3.1 Le générateur de tension

1.3.1.1 Le système de charge

Le système de charge [6] est constitué essentiellement d'une courroie de 100 mètres de long tournant à une vitesse plus faible (~10 m/s) que dans les systèmes existants (21 m/s pour un Tandem MP de 15 MV). Il délivre un modeste courant de 500 µA, comparé aux 700 µA (2 x 350) qui peuvent être obtenus sur un Tandem MP de 15 MV. La particularité de ce système est que la courroie traverse les deux étages d'accélération de la machine. Il permet surtout d'atténuer les vibrations en diminuant la vitesse de rotation de moitié. C'est la seule machine au monde où la courroie se déplace d'une extrémité à l'autre de la cuve.

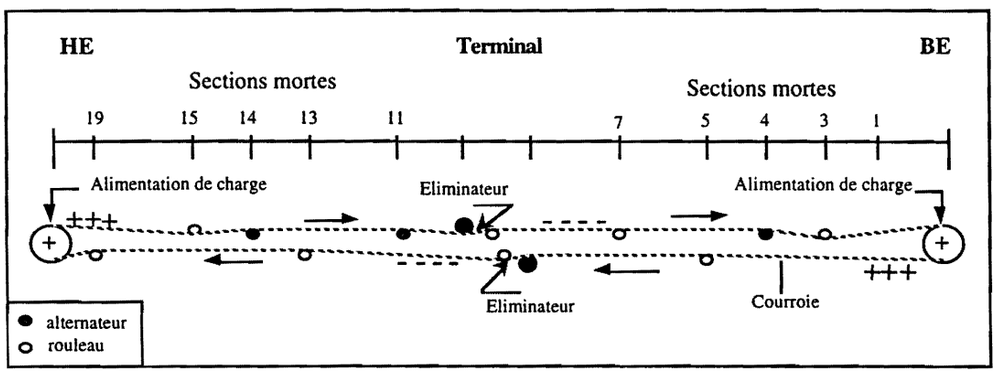


Figure A.7 - Emplacement des équipements du système de charge

La figure A.6 nous montre l'emplacement des différents équipements du système de charge. Deux moteurs, un en HE et l'autre en BE, assurent l'entraînement de la courroie qui, à son tour, met en rotation les alternateurs. Deux alternateurs en SM4 et SM14 respectivement alimentent les pompes ioniques et deux alternateurs au terminal alimentent les pompes cryogéniques. La case d'équipement en SM11 est alimentée par un petit alternateur (200 W).

Une alimentation de charge en BE (15 kV) dépose les charges sur la courroie. Celles-ci sont récupérées au niveau du terminal par un peigne éliminateur de charges. Le même système est utilisé côté HE.

La mesure des courants colonne, tube et éliminateurs donne des informations importantes sur l'état de la machine et permet notamment de calculer le potentiel électrique du terminal.

1.3.1.2 Le tube accélérateur

Dans les accélérateurs Tandem classiques, le tube repose dans une colonne qui tient toute seule par simple pression sur les extrémités, alors que dans le Vivitron (figure A.5), de par ses dimensions (deux fois plus long), la colonne doit être soutenue mécaniquement par des empilements de plots isolants en époxy de 40 cm de long, pouvant supporter une différence de potentiel de 5 MV chacun. Le tube est composé d'une électrode terminale de 1,4 mètres de diamètre, située au centre du réservoir et de 8 sections accélératrices, de part et d'autre, supportant chacune 5 MV et séparées par des sections mortes où le champ est nul. Il est entouré par une structure colonne dont les éléments sont des planches isolantes de 2,8 mètres de longueur, non subdivisées, en fibre de verre et résine époxy.

Cette structure est à son tour enveloppée par 48 électrodes colonne de chaque côté du terminal, associées à une chaîne de résistances qui assurent une répartition longitudinale du potentiel. Ces électrodes, de par leur forme légèrement inclinée, protègent électriquement la structure et le tube en jouant le rôle d'éclateurs. L'utilisation de la fibre de verre et de la résine époxy allège considérablement le poids de la partie longitudinale de la structure (réduction d'un facteur 10) et a permis la construction d'une machine horizontale.

1.3.1.3 Les électrodes discrètes

Sur une machine classique, le champ électrique varie en $1/r$ où r est la distance entre l'électrode centrale et le réservoir. Dans le Vivitron, l'introduction de 7 électrodes discrètes assemblées en portiques entre le tube et l'enceinte crée un champ électrique uniforme. Ces portiques sont reliés entre eux par des plots isolants qui assurent la stabilité longitudinale de l'ensemble et encaissent les forces axiales introduites par leur inclinaison. L'utilisation des électrodes discrètes a permis de réduire considérablement le diamètre central de l'accélérateur par rapport à une conception classique, pour une même tension et un même champ électrique.

1.3.2 L'accélérateur d'ions

1.3.2.1 Les sources d'ions

Trois méthodes sont utilisées pour la production d'ions négatifs : l'échange de charge (*Duoplasmatron*), l'extraction directe (*Penning*) et le bombardement d'une cathode par un faisceau de césium positif (*Sputtering*) [5]. L'injecteur comprend deux voies analysées en masse par un dipôle

magnétique de ±35° avec un pouvoir séparateur de $\Delta m/m$ = 1/20, et une voie centrale, appelée troisième voie (figure A.7), analysée en masse par un dipôle magnétique de 75° qui donne une meilleure qualité optique avec un pouvoir séparateur de $\Delta m/m$ = 1/200.

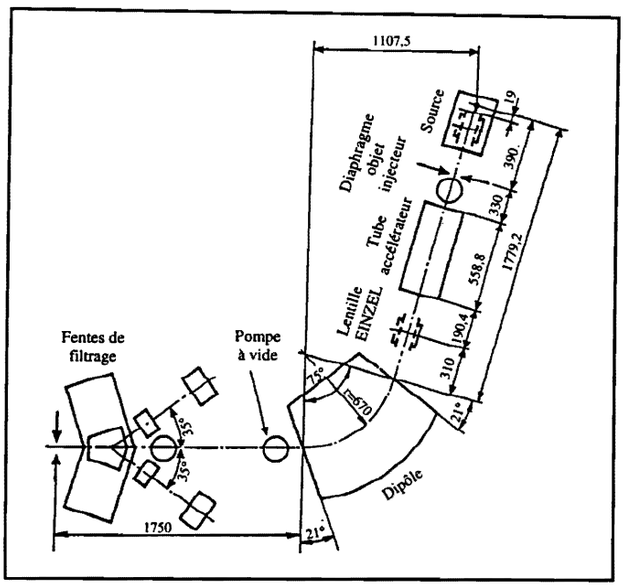


Figure A.7 - La troisième voie

Le faisceau émis par les sources 1 et 2, placées à un potentiel de 20 kV par rapport à la plate-forme (300 kV), est analysé en masse par le dipôle magnétique 35° et focalisé sur l'iris avant d'être préaccéléré par le tube 300 kV. Le faisceau émis par la source 3 (troisième voie), placée à un potentiel de 80 kV par rapport à la plate forme, est refocalisé par une lentille électrostatique en un point représentant le point objet de l'injecteur. Une cage de Faraday (*Faraday cup*) et un profileur permettent ce réglage. Les ions sont ensuite accélérés par le tube 60 kV, déviés et analysés en masse par le dipôle magnétique 75° et finalement focalisés par un *steerer* (lentille Einzel) sur les lèvres d'analyse (diaphragme IRIS2 (figure A.8)). Après avoir coupé l'alimentation de l'aimant 35°, le faisceau est préaccéléré par le tube d'accélération 300 kV.

L'énergie d'injection à l'entrée de l'accélérateur est de 320 keV pour les sources 1 et 2 et de 380 keV pour la source 3. Cette énergie est composée par 300 keV provenant du tube de préaccélération, 20 keV provenant de la tension d'extraction de la source et, pour la source centrale, 60 keV provenant du tube 60 kV.

Nous ferons dans ce paragraphe un bref commentaire sur la pulsation d'ions. Le principe est de recueillir un maximum d'ions dans un intervalle de temps minimum et ne laisser injecter dans la machine qu'un paquet sur 2, 3... (on veut des impulsions inférieures à la nanoseconde [7]). La pulsation est assurée en BE par un groupeur à double glissement, à fréquence variable, associé à un hacheur à plaques distribuées.

1.3.2.2 Le faisceau

A la sortie du tube d'accélération 300 kV, le faisceau d'ions est refocalisé par un *steerer* composé d'une lentille électrostatique ESAG (*ElectroStatic Alternating Gradient*) vers le point objet de l'accélérateur. Ensuite, il est propulsé par l'ensemble hacheurgroupeur et diagnostiqué autour du

point objet par un émittance-mètre et une boite de diagnostic regroupant le diaphragme IRIS3, un profileur et une cage de Faraday. Le faisceau doit être injecté dans la machine légèrement décentré et incliné verticalement pour réduire l'amplitude de la première oscillation due aux électrodes inclinées [5]. Les *steerers* verticaux en BE sont prévus à cet effet.

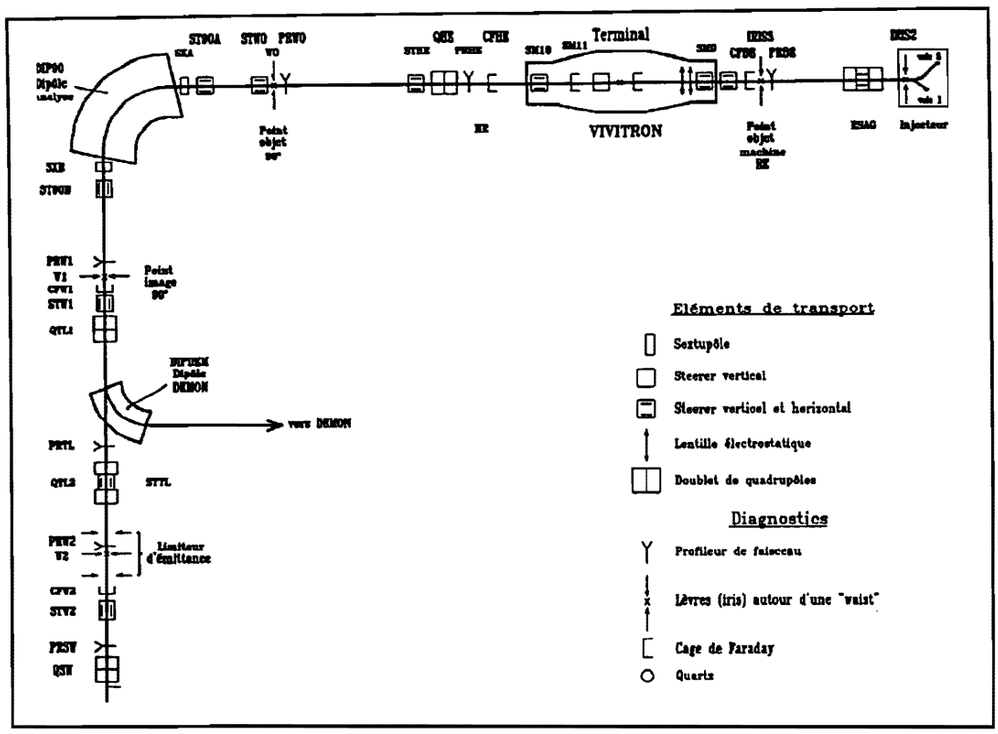


Figure A.8 - Les principaux éléments de transport et diagnostic du faisceau d'ions

Le faisceau divergent est ensuite refocalisé dans le canal de l'éplucheur à feuilles de carbone (stripper feuilles) par une lentille grille en SM0. Pour les atomes lourds, un casse-molécule ou éplucheur à gaz (*stripper* gaz) est associé au stripper feuilles.

Un sélecteur de charge, composé d'un steerer vertical au terminal et d'un steerer vertical-horizontal en SM18, refocalise le faisceau sur le post-éplucheur et disperse les différents états de charge pour qu'un seul état soit accéléré en HE.

A la sortie de l'accélérateur, le faisceau divergent est refocalisé par un doublet quadripolaire magnétique pour passer entre les lèvres du dipôle 90° qui l'analyse en masse et en charge. Pour pouvoir analyser le faisceau en différents points de la machine, une boite de diagnostic a été placée sur le point objet de l'aimant 90° et une cage de Faraday en SM11.

1.3.3 Le vide

Avant de présenter le vide dans le Vivitron nous allons rappeler quelques définitions importantes pour la suite de l'exposé :

- **Pression partielle**
  C'est la pression qu'un gaz ou une vapeur aurait s'il se trouvait isolé dans un réservoir.
- **Pression totale**
  C'est la somme des pressions partielles de tous les gaz et vapeurs présents dans un réservoir.

Dans l'atmosphère, la pression varie en fonction de l'altitude à laquelle on se trouve, la pression atmosphérique de référence étant de l'ordre de 1013 mbar (altitude zéro). C'est grâce à des paliers de pression inférieure à la pression de référence que l'on définit les domaines du vide :

- **vide grossier** --> 1013 à 1 mbar ;
- **vide moyen** --> 1 à 10^{-3} mbar ;
- **vide élevé** --> 10^{-3} à 10^{-7} mbar ;
- **ultra-vide** --> pression inférieure à 10^{-7} mbar.

Dans une machine comme le Vivitron, le vide est principalement présent dans le tube accélérateur, sa répartition se faisant en fonction de l'emplacement des pompes (figure A.9). Le vide au terminal est réalisé par deux pompes cryogéniques (débit : 100 l/s) et entretenu par deux pompes ioniques, l'une sur la SM4 et l'autre sur la SM14, qui ne sont actionnées qu'à partir de 10^{-4} T. Le vide dans le tube accélérateur est assuré par deux groupes de pompage, l'un en BE à l'entrée de la machine et l'autre en HE à la sortie de la machine. Ces deux groupes sont composés d'une pompe secondaire turbo-moléculaire et d'une pompe primaire à palettes.

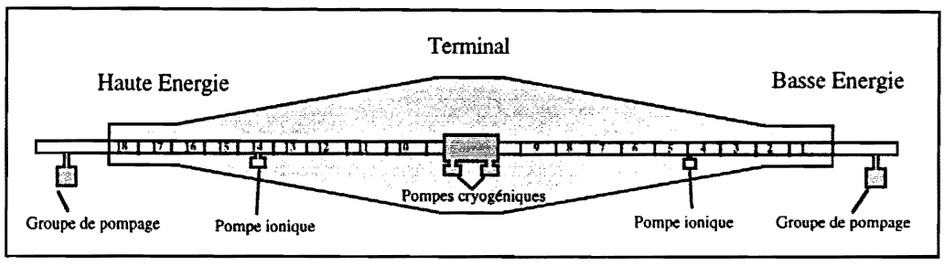


Figure A.9 - Localisation des pompes à vide dans la cuve

Avant de pressuriser le réservoir avec du SF_6, on établit un pré-vide (~ 10^{-2} mbar) pour obtenir une plus grande concentration du gaz lorsqu'il est injecté dans la machine. Ce pré-vide est créé par un groupe de pompage situé sous la machine. Deux groupes de pompage sont installés en aval et en amont de l'aimant 75° de l'injecteur pour assurer le vide lors de l'utilisation des sources Duoplasmatron, lesquelles utilisent une grande quantité de gaz.

Le vide est mesuré par deux types de jauges de pression : une jauge Pirani pour un vide de 100 à 10^{-3} mbar et une jauge à cathode froide (*Penning*) pour le vide élevé et l'ultravide de 5.10^{-3} à environ 10^{-11} mbar. Le vide est préservé à l'intérieur du tube d'accélération de la machine et des tubes de préaccélération de l'injecteur par des vannes de sécurité situées à l'entrée et à la sortie de la machine, d'une part, et aux différentes extrémités des différents tubes de préaccélération de l'injecteur, d'autre part.

La figure A.10 nous montre le principe d'un groupe de pompage. Les deux vannes VI et VII sont fermées. La pompe primaire à palettes est mise en route et aspire le gaz qui se trouve dans le bas de la pompe turbomoléculaire jusqu'à atteindre une pression d'environ 5.10^{-3} mbar. Cette opération dure à peu près deux minutes. La turbine de la pompe à vide élevé commence ensuite à tourner et la vanne VII s'ouvre pour pomper le volume de la cuve et arriver à une pression d'environ 10^{-7} mbar.

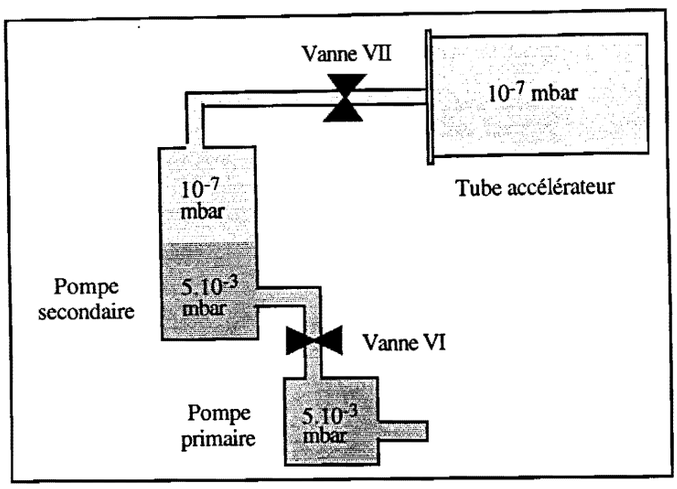


Figure A.10 - Principe d'un groupe de pompage

1.4 Les paramètres à contrôler

1.4.1 Leur localisation

La mise en route d'un accélérateur électrostatique comme le Vivitron demande un système de contrôle et commande particulier. En effet, les paramètres à commander et à contrôler sont distribués dans un périmètre géographique étendu, dont certains sont localisés à l'intérieur de la cuve (tension variant de 0 à 35 MV) et de l'injecteur (tension variant de 0 à 380 kV). Le tableau T1 nous montre la répartition des équipements suivant leur localisation.

Localisation	Nombre d'équipements	Nombre de lectures	Nombre de commandes	pourcentage
Injecteur	106	290	145	25 %
Réservoir Interne	172	305	94	41 %
Extension BE	47	163	115	11 %
Réservoir Externe	56	130	54	13 %
Extension HE.	21	38	23	5 %
Analyseur	21	67	61	5 %
Ligne Faisceau	Installation à prévoir			
Total	~ 423	~ 993	~ 492	
Total paramètres	~1495			

Tableau T.1 - Répartition des équipements suivant leur localisation

La gestion des paramètres situés à l'intérieur de la cuve, dans le terminal et dans les zones équipotentielles des sections mortes est indispensable pour le fonctionnement du Vivitron. Les équipements embarqués qui doivent gérer ces paramètres sont soumis aux contraintes physiques d'un environnement extrêmement sévère.

1.4.2 Les problèmes rencontrés

La protection des équipements électroniques est particulièrement difficile [8]. La conception de cette protection doit, entre autres, supporter les perturbations électromagnétiques provenant essentiellement des décharges de 35 MV dont l'énergie totale ($W = 1/2\ CV^2$) peut théoriquement atteindre 800 kJ [9]. Notons que dans les accélérateurs Tandem traditionnels la quantité d'énergie emmagasinée est estimée à 60 kJ pour 13 MV et 114 kJ pour 18 MV (figure A.11).

De plus, dans un système à électrodes discrètes comme celui du Vivitron, les charges électriques sont réparties sur une grande étendue, contrairement aux accélérateurs Tandem classiques où tout est concentré au terminal. Des perturbations électriques peuvent également provenir de lignes équipotentielles du champ électrostatique qui pénètrent dans les sections mortes.

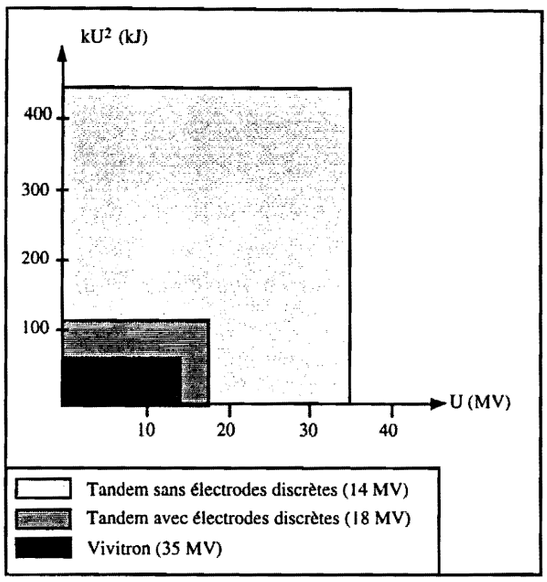


Figure A.11 - Quelques estimations d'énergie stockée

Cette conception doit aussi prendre en compte les problèmes posés par les rayonnements ionisants et les contraintes de l'environnement physique qui correspondent au vide, à la surpression gazeuse de SF_6 et aux perturbations électromagnétiques. La solution retenue consiste à abriter les équipements dans des caissons gigognes réalisant un double blindage, électrostatique et électromagnétique, et constituant les cases d'équipements. Les équipements électroniques doivent résister au vide, à la surpression gazeuse et aux perturbations électromagnétiques. Les premiers essais ont montré un comportement correct en tension, aux claquages et sous SF_6. Ce gaz est caractérisé par une meilleure isolation à la haute tension et par une bonne dissipation thermique.

2 Critères de choix sur le processus

La mise en place d'un système de contrôle et commande pour gérer une machine comme le Vivitron n'est pas triviale. Près de 1500 paramètres doivent êtres contrôlés et commandés sur une surface approximative de 2400 m^2. Toute l'information fournie par les variables physiques est centralisée sur un pupitre de contrôle situé à l'extérieur du hangar du Vivitron.

Imaginons un instant les kilomètres de câble qui seraient nécessaires pour transporter toutes les informations utiles jusqu'au pupitre, la complexité de ce câblage, son prix de revient, etc. L'arrivée de torons de câbles au pupitre obligerait à protéger cette salle contre les claquages intempestifs. Un nombre élevé de paramètres devraient être visualisés sur des panneaux de contrôle immenses et, lors du réglage de certains paramètres, il faudrait plusieurs mains pour tourner tous les boutons et plusieurs yeux pour surveiller toutes les informations sur les différents panneaux de contrôle. Si nous voulions conserver un historique ou les traces de comportement, où et comment pourrions-nous stocker les valeurs ?

Les mesures et les commandes à effectuer au niveau de l'électrode terminale (les mesures du vide ou les commandes des pompes, par exemple) impliquent la mise en place de plusieurs cases d'équipements à cet endroit. Pour communiquer depuis le potentiel de la terre à cette électrode, pouvant être portée à une tension maximale de 35 MV, une liaison de type optique s'impose. De plus, pour vérifier l'étanchéité des plots, lors des premiers tests de la machine, il fallait mesurer les courants de fuite. Ceux-ci ne pouvaient être mesurés que par des capteurs de courant nécessitant à nouveau un signal optique pour décoder le signal numérique des capteurs.

La souplesse et la fonctionnalité d'un système informatisé s'imposent donc, pour le contrôle et la commande d'une machine comme le Vivitron. Des études ont été menées dans trois directions différentes : la connaissance du processus, la recherche de solutions aux problèmes techniques spécifiques et la connaissance des solutions des autres laboratoires [10].

2.1 L'analyse du système de contrôle et commande

2.1.1 La connaissance du processus

Pour arriver à bien définir le processus, il a fallu recenser toutes les mesures et les commandes et décrire en détail tous les équipements nécessaires à celui-ci. Un dossier d'équipements a été ainsi réalisé pour présenter les moyens (de liaison, logiciels, logistiques, humains et financiers) et les buts recherchés :

- établir le catalogue des spécifications techniques à prendre en compte ;
- définir la localisation et la nature des capteurs et des actionneurs ;
- définir les caractéristiques statiques et dynamiques des signaux ;
- évaluer le nombre de paramètres à gérer ;
- définir les paramètres et les lois qui les régissent.

2.1.2 La recherche de solutions aux problèmes techniques spécifiques

Nous avons mis en place une série de tests sur le comportement et les liaisons des équipements. Ces tests, qui concernaient surtout les équipements situés dans la cuve (§1.4.2), ont été effectués en temps utile sur l'accélérateur MP de Strasbourg.

2.1.3 La connaissance des solutions des autres laboratoires

Le projet Vivitron a été élaboré en collaboration avec les différents constructeurs associés et avec les experts internationaux des différents domaines. Cependant, l'absence de références fiables due au nombre très limité, au niveau mondial, de machines semblables au Vivitron a souvent ralenti l'avancement du projet.

Après avoir caractérisé chaque équipement, déterminé le nombre d'équipements nécessaires et le nombre de paramètres correspondant aux variables physiques, nous avons pu commencer la réalisation d'un système de contrôle et commande sur la base de trois critères : la vitesse, la fonctionnalité et la fiabilité.

2.2 La vitesse

Un système de contrôle, s'il ne suit pas ses processus, ne peut pas les contrôler. Nous pouvons dire « *qu'un système fonctionne en temps réel à chaque fois qu'il sera question de contraintes de temps et que ces dernières seront respectées* » [11]. Prenons comme exemple l'interface opérateur du système de contrôle du Vivitron. L'affichage des paramètres sur les écrans doit se faire à l'échelle humaine. Le temps de réponse d'une commande ne doit pas être trop lent afin que l'opérateur ne soit pas troublé et ne répète pas la commande. La lecture d'une mesure ne doit pas être trop rapide afin que l'opérateur ait le temps de visualiser la variation du paramètre. Donc l'interface opérateur n'a pas besoin d'être très rapide. En revanche, des traitements informatiques nécessitent une certaine rapidité comme la lecture simultanée de tous les paramètres de mesure (~ 1000). Mais là encore, un affichage graphique informatisé peut gérer de l'ordre de 100 mesures/seconde, ce qui est suffisant si l'on sait qu'une grande partie des mesures maintiennent une valeur constante pendant quelques minutes, une fois la machine stabilisée.

Calculons maintenant le temps de réponse du Vivitron. Deux alimentations de charge, l'une en HE et l'autre en BE, avec un temps de réponse de 200 ms, commencent à déposer des charges sur une courroie qui tourne à une vitesse de 10 m/s. Ces charges parcourent une distance de 25 mètres en quelques secondes avant d'être récoltées au terminal. Le temps nécessaire pour que toutes les charges soient récupérées au terminal et que le champ électrique devienne alors uniforme sur l'ensemble de la machine est estimé à environ 25 minutes. Pour le phénomène le plus rapide, l'effet corona utilisé pour la stabilisation en tension, le temps de réponse est de l'ordre de 80 ms (à vérifier par la mesure). Nous remarquons que le temps de réponse varie considérablement suivant le domaine d'application :

- la centaine de milliseconde pour la stabilisation en tension ;
- quelques centaines de millisecondes pour les alimentations de charge ;
- la seconde pour les systèmes de visualisation humaine ;
- environ 25 minutes pour le Vivitron.

Pourtant ces quelques chiffres montrent bien que le Vivitron est une machine relativement lente. D'autres appareils, comme les vannes de pression, n'ont pas besoin d'une surveillance rapide. Tous ces phénomènes sont très lents et ne demandent pas un contrôle informatique extrêmement rapide.

Dans notre cas, la notion de **vitesse** n'est pas attachée à la notion de **rapidité** dans le sens où les processus du Vivitron ne sont pas généralement rapides. Il suffit que **les contraintes de temps soient respectées** et donc que la vitesse du système s'adapte au processus à contrôler (par exemple, la période de la courroie est connue en prenant un échantillon toutes les 500 µs).

2.3 La fonctionnalité

La **fonctionnalité** est, sans doute, le critère le plus lié à l'informatisation du système. Voir et contrôler tous les paramètres sur quelques écrans réduit considérablement l'espace d'action de l'opérateur. En cas d'alarme mineure, l'informatique peut prendre des initiatives pour agir au niveau matériel. Cette action peut être représentée de façon à attirer l'attention de l'opérateur sur un paramètre précis (clignotement de l'affichage, modification de la couleur, etc.). L'**ergonomie** de l'interface opérateur (utilisation de la souris, écrans graphiques, etc.) et les **outils conviviaux** de l'informatique rendent la tâche plus agréable et motivent davantage l'opérateur.

La possibilité d'avoir un **historique** est importante pour connaître l'état des processus avant et après une défaillance (panne, alarme, mauvaise manipulation, etc.). Le choix de reproduire une configuration de la machine est ainsi possible.

2.4 La fiabilité

La première caractéristique essentielle d'un système temps réel doit être sa **fiabilité** et sa **sûreté**. Deux paramètres sont à considérer : la fréquence des défaillances et leurs conséquences sur le fonctionnement du système. Il est important de minimiser l'effet de défaillance en exigeant que l'arrêt d'un élément du système ne doive pas se traduire par l'arrêt de tout le système. Un fonctionnement en mode dégradé doit assurer la continuité du système.

Cependant, la dégradation d'un système est autant liée aux contraintes matérielles qu'aux contraintes logicielles. Prenons un exemple : un satellite utilise des programmes téléchargeables parce que le logiciel n'est pas fiable. Mais, si le matériel tombe en panne, le satellite n'est plus sous contrôle. Un programme informatique, une fois mis au point, ne peut plus être défaillant dans l'application pour laquelle il a été conçu. Le développement informatique ne peut être lié à son exploitation : une version non encore testée peut se glisser dans la version d'exploitation et rompre la fiabilité du système. Un programme doit être débarrassé de tous ses défauts, mais cela ne peut être garanti. Les difficultés de test en vrai grandeur, comme dans notre projet Vivitron, ne font que confirmer cette affirmation.

Le comportement d'un opérateur face à une situation critique n'est pas toujours prévisible. Cependant, nous avons choisi de ne faire appel qu'à des **dispositifs matériels** quand il s'agit de **sécurités vitales** pour le personnel et le matériel, et au **logiciel** quand il s'agit de **sécurités mineures** (§2.3). Par exemple, la sécurité de l'homme dans les lieux sensibles comme l'injecteur ne doit pas être prise en compte par l'informatique mais par des protections matérielles (sirènes, gyrophares, portes fermées, coupure du faisceau, etc.) car cela nécessite des machines tolérantes aux fautes des systèmes redondants dont le coût est prohibitif.

Nous avons aussi choisi des **dispositifs matériel/logiciel** tels que les **chiens de garde** ou *watchdog timers* (chapitre B, §3.1.2.3) pour redémarrer notamment les ordinateurs frontaux qui se trouvent dans des secteurs critiques tels que l'injecteur ou l'intérieur de la cuve. Si nous perdons la communication avec un l'ordinateur frontal du terminal, il faut intervenir à l'intérieur de la cuve pour le redémarrer, ce qui implique l' ouverture de la machine et un arrêt de 53 heures environ .

Séquence	Opération
Ouverture	Extraction SF_6 (15 heures) Admission d'air (2 heures) Ventilation (2 heures)
Fermeture	Extraction de l'air (14 heures) remplissage de SF_6 (20 heures)

Tableau T.2 - Durée des différentes opération d'ouverture et de fermeture du Vivitron

Des handicaps connus forment ce qu'on peut appeler la "non fiabilité". Plongée dans le SF_6, l'électronique supporte une température normale mais, à l'air sec (gaz moins dense et donc mauvais caloporteur), des problèmes de capacité calorifique se posent et les équipements électroniques ont tendance à surchauffer. Si un équipement interne à la machine tombe en panne et doit être changé, il faut arrêter la machine pendant 53 heures (tableau T.2).

Les équipements électroniques situés à l'intérieur de la cuve doivent être protégés (double blindage, parasurtenseurs) des conditions physiques très rigoureuses (§1.4.2). Les liaisons avec le monde extérieur ne peuvent être que de type optique à cause des hautes tensions. L'utilisation d'un bus de terrain (BITBUS, FIELBUS, CAN, etc.) empêche de mettre l'intelligence dont nous aurions la maîtrise totale au plus près des équipements et oblige à prendre un type spécifique de matériel, pas toujours adapté aux caractéristiques du Vivitron. Le choix du matériel doit donc être fait sur des produits standards, déjà testés et fiables. Les équipements tant à l'intérieur qu'à l'extérieur de la cuve doivent être protégés des perturbations du secteur par un onduleur.

Enfin, pour avoir un système parfaitement fiable, il faudrait avoir la possibilité de remplacer le matériel dit "irremplaçable" dès qu'il est défaillant. Au niveau logiciel, on a toujours la possibilité de configurer le système de façon à éviter les appareils "irremplaçables". Les techniques de redondance tendent à favoriser un fonctionnement fiable et sûr mais, en contrepartie, elles dégradent les performances temporelles de l'ensemble du système. Au niveau matériel, il faudrait dédoubler certains équipements, mais ceci est d'un coût prohibitif et rarement possible. Prenons comme exemple la mesure de l'altitude sur un Airbus. Pour fiabiliser au maximum cette mesure, le système est composé de 3 altimètres qui se contrôlent mutuellement. Chaque altimètre possède son Unité de Processeur Centrale (CPU), son système opératoire (OS) et son propre programme. Pour mesurer l'altitude, l'Airbus utilise donc 3 CPUs, 3 OS et 3 programmes différents.

— B —

Le Système de Contrôle et Commande du Projet Vivitron

1 Généralités sur les communications dans un environnement réparti UNIX

1.1 Le modèle de référence OSI

En 1978, pour éviter la multiplication des solutions d'interconnexion de systèmes basés sur des architectures hétérogènes, un organisme international, l'ISO (*International Standard Organisation*) a développé un modèle de référence appelé OSI [12] (*Open System Interconnection*). Il permet l'interconnexion de systèmes d'origines différentes dans le respect des normes et des protocoles de ce modèle. Les systèmes qui sont conformes à ces conventions sont dits **ouverts**. Le modèle OSI ne concerne pas l'architecture interne des systèmes, mais leur comportement externe permettant la communication.

Ce modèle d'architecture en couches permet d'abaisser le niveau de complexité de la communication entre deux systèmes informatiques d'origines différentes en décomposant cette communication en un certain nombre de problèmes plus simples à traiter. C'est un modèle avec une approche descendante (*top-down*) et un raffinement graduel.

1.1.1 Le Modèle en couches

Le premier à avoir mis sur pied une architecture de communication utilisant une structure en couches est IBM en 1973 avec SNA (*System Network Architecture*). Les différentes fonctionnalités sont rangées dans des couches séparées interagissant entre elles : la couche d'un niveau ***n*** rend un service à la couche supérieure d'un niveau ***n +1***, qui elle-même rendra un service à la couche d'un niveau ***n +2***, la couche ***n*** ne pouvant pas rendre directement un service à la couche ***n +2***.

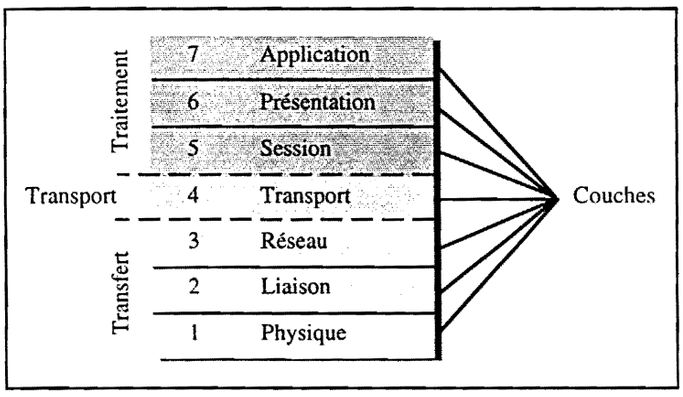


Figure B.1 - Les sept couches du modèle OSI

Tous les concepts modernes de la communication sont déjà présents dans cette architecture. L'ISO, poussé par la puissance des concepts de SNA, va proposer une architecture de communication en

couches, normalisée, bâtie autour du modèle OSI. Ce modèle se compose de 7 couches (figure B.1) divisées en deux ensembles : transfert de l'information (couches 1, 2, 3) et traitement de l'information (couches 5, 6, 7) avec la couche 4 qui fait charnière entre les deux.

- **La couche physique**
  Elle assure la transmission d'un train de bits sur un média physique (paire torsadée, câble coaxial, fibre optique). Cette couche définit les caractéristiques du média, des encodeurs, des émetteurs/récepteurs (*transceivers*), des boîtiers de connexion, du câblage en général, des signaux, etc.

- **La couche liaison**
  Elle est responsable de l'acheminement sans erreur de blocs d'information (trames) sur une liaison physique. Elle détecte et corrige les erreurs induites par la couche physique. Comme les transmissions ne sont pas fiables, la couche liaison définit un mécanisme d'échange d'accusés de réception qui permet de savoir si une trame a été correctement transmise. Cette couche est divisée en deux parties : la sous-couche **MAC** (*Medium Access Control*), qui définit la méthode d'accès propre à chaque type de support physique, et la sous-couche **LLC** (*Logical Link Control*), commune à tous les types de supports physiques et de méthodes d'accès, masquant ainsi ses spécificités aux couches supérieures.

- **La couche réseau**
  Elle définit les protocoles capables d'acheminer des paquets de données au travers d'un ou plusieurs nœuds de communication intermédiaires. Elle assure l'établissement, le maintien et la libération des connexions entre systèmes, ainsi que l'adressage, le routage et le contrôle de flux. Elle offre deux types de services :
  - Le circuit virtuel ou **mode connecté** (*connection-oriented*). Une connexion virtuelle est établie entre les deux machines et le transfert des données est garanti.
  - Le datagramme ou **mode non connecté** (*connectionless*). Le transfert des données n'est pas garanti.

- **La couche transport**
  C'est une couche charnière entre le transport physique des données et le traitement de l'information (couches 5, 6, 7). Elle est responsable du contrôle du transport des informations de *bout en bout*, au travers du réseau et garantit aux couches supérieures une qualité de service constante pour le transfert des données.

- **La couche session**
  Elle assure la synchronisation et le contrôle, par logiciel, du dialogue entre les différents systèmes.

- **La couche présentation**
  Elle permet aux systèmes qui échangent des données d'interpréter celles-ci indépendamment de leur représentation syntaxique dans ces systèmes. Elle prépare l'information en un format compréhensible pour la couche application.

- **La couche application**
  Elle fournit à l'utilisateur d'un système ouvert (opérateur, programme d'application, etc.) des services de haut niveau (messagerie électronique...).

Considérons deux systèmes A et B (figure B.2). Dans un même système, chaque couche n'interagit qu'avec ses deux couches adjacentes par des **Unités de données de Service** S D U

(*Service Data Units*) : la couche ***n*** rend un service à la couche ***n + 1*** en s'appuyant sur les services de la couche ***n - 1***. Entre les deux systèmes, la couche ***n*** du système A dialogue avec la couche ***n*** du système B, de même rang (couche **correspondante**), en échangeant des **Unités de Données de Protocole** PDU (*Protocol Data Units*).

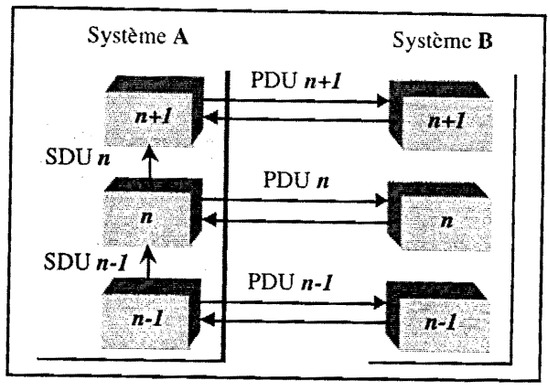


Figure B.2 - Les protocoles en couche

Le mécanisme de demande de service à la couche inférieure utilise deux modes de service de connexion : le mode *connecté* et le mode *non connecté*. Le mode *non connecté* ne fait appel à aucun mécanisme pour établir la connexion entre deux couches adjacentes. L'échange de PDU de données de ***n + 1*** à destination de ***n*** se fera librement. En mode *connecté*, il y a une mise en relation préalable des couches qui ont un échange d'information. Donc une requête d'ouverture de connexion au niveau ***n*** est toujours émise à partir du niveau ***n + 1***. A une demande de connexion doit toujours correspondre une demande de déconnexion.

1.2 Les protocoles TCP/IP ou Internet

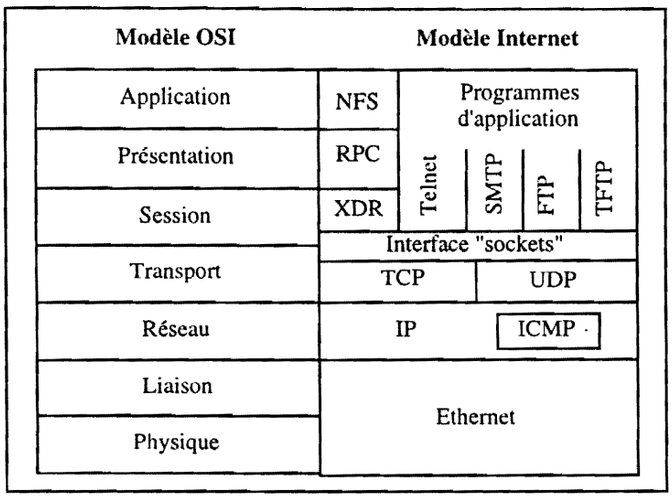


Figure B.3 - Les protocoles Internet

1.2.1 Introduction

L'étude de l'interconnexion de réseaux a débuté au début des années 1970, parce que l'on souhaitait qu'un ordinateur connecté à un réseau puisse communiquer, sans s'occuper du type des réseaux ni du nombre de ceux-ci, avec n'importe quel autre ordinateur de n'importe quel autre réseau. Les travaux de la Direction américaine des recherches DARPA (*Defense Advanced Research Projects Agency*) aboutirent à l'apparition d'un réseau virtuel unique appelé Internet avec une première utilisation des protocoles TCP/IP (*Transmission Control Protocol / Internet Protocol*). Ces protocoles définissent les règles utilisées pour échanger les unités de données, les détails de leur structure et indiquent comment gérer les erreurs.

1.2.2 Le protocole IP

Le protocole IP (*Internet Protocol*) [13] correspond à la couche 3 du modèle OSI (figure B.3) et offre les fonctions de base nécessaires à l'interconnexion de systèmes ouverts. Il fonctionne en *mode non connecté*, ce qui n'assure pas un transfert fiable des messages. La perte de messages, un mauvais ordre d'arrivée ou une modification des données durant le transfert ne sont pas détectés. Il n'y a pas de contrôle de flux ni de procédure de retransmission. Ces deux fonctionnalités sont repoussées vers le protocole de niveau supérieur, TCP, qui effectue une transmission fiable des messages en *mode connecté*. Certaines fonctions de contrôle sont toutefois assurées par un protocole annexe de IP : ICMP (*Internet Control Message Protocol*).

TCP considère le **flot de données** (*streams*) comme une séquence d'octets qu'il divise en paquets appelés segments. Ces segments, transférés par IP en *mode non connecté*, deviennent des **datagrammes** pour être acheminés de sous-réseau en sous-réseau par les différents nœuds intermédiaires, jusqu'au destinataire. Notons qu'un datagramme est l'unité de transfert de base d'une interconnexion TCP/IP. Il possède un en-tête, contenant les adresses IP source et destination, et une zone de données servant à acheminer tout type de données. Un datagramme IP est construit avec une taille maximum dépendant du réseau initial, par exemple 1500 octets sur Ethernet.

1.2.2.1 Structure d'adressage

Chaque station dispose d'une adresse IP unique (appelée aussi adresse Internet) codée sur 32 bits et constituée d'un numéro de réseau et d'un identificateur de machine. A ces adresses sont associés des noms symboliques (*hostname*). Le **numéro de réseau** identifie le réseau physique et est attribué de façon unique par une autorité spécifique : le NIC (*Network Information Center*). L'identificateur de machine identifie la machine sur ce réseau ; il peut lui-même être divisé en deux parties : l'une pour identifier un sous-réseau et l'autre pour la machine elle-même. La configuration du réseau et l'attribution des adresses IP et des *hostnames* sont normalement assurées par l'administrateur du système ou du réseau par l'intermédiaire, en général, du service NIS (*Network Information Service*).

En fonction de la taille du réseau (en nombre de sous-réseaux) et du nombre de machines connectables par sous-réseau, trois classes d'adresses sont proposées (figure B.4). La classe étant définie par les trois bits de poids fort.

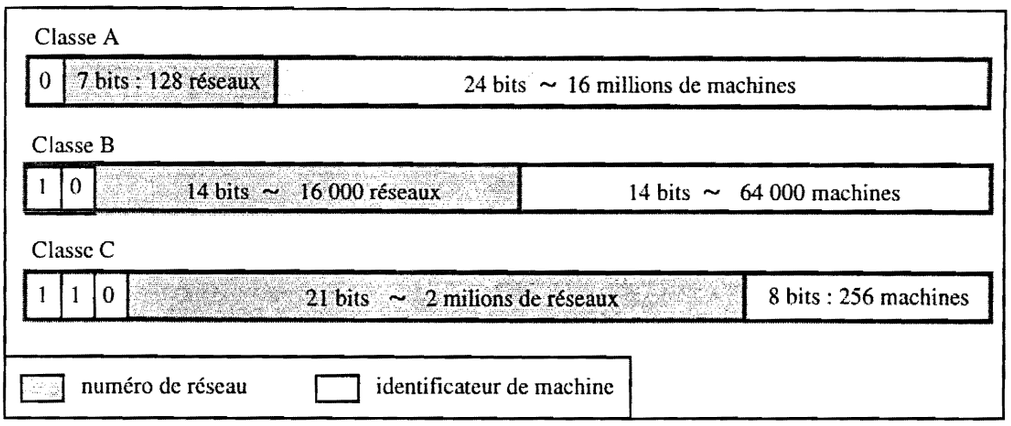


Figure B.4 - Structure d'adressage IP

Une adresse IP est formée de 4 octets écrits en représentation décimale pointée : A.B.C.D. Le CRN de Cronenbourg utilise des adresses de classe C (exemple : 193.48.88.210). Par convention [13], un identificateur de machine où tous les bits sont à 0 ou à 1 n'est jamais affecté à une machine. Ses valeurs sont réservées à des adressages particuliers :

- **Adresse du réseau**
  Tous les bits de l'identificateur de machine sont nuls.
  Cette adresse désigne le réseau sur lequel se trouve la machine. Les réseaux qui concernent le projet Vivitron sont le réseau *Vivitron* (193.48.88.0) et le réseau *Acquisition de Données* (193.48.87.0).
- **Adresse de diffusion** (*broadcast address*)
  Tous les bits de l'identificateur de machine sont à 1.
  Cette adresse désigne une diffusion du message sur toutes les machines du réseau.

1.2.2.2 Le routage des datagrammes IP

Les interconnexions sont constituées d'un ensemble de réseaux physiques reliés par des machines appelées **passerelles** (*gateways*). Une passerelle est directement connectée à deux ou plusieurs réseaux contrairement à une machine qui, elle, est reliée directement à un seul réseau physique. Une passerelle est aussi appelée **routeur** car elle a pour fonction de choisir une route suivant laquelle les datagrammes vont être transmis. Le **routage IP** est l'ensemble des processus qui permettent la détermination de la prochaine passerelle. La route est directe si la machine destination et la machine émettrice se trouvent sur le même réseau physique. L'émetteur encapsule le datagramme dans une trame physique, résout l'association **adresse IP / adresse physique** et envoie directement la trame au destinataire. La route est indirecte si le datagramme doit transiter par une passerelle afin d'être acheminé vers un autre réseau.

Les datagrammes sont routés vers la passerelle la plus proche. Ils transitent de passerelle en passerelle jusqu'à ce que l'une d'entre elles puisse les remettre directement à son destinataire. Si sur leur chemin ils doivent emprunter un réseau plus restrictif (forte activité...), la passerelle les découpe en fragments. Seul le destinataire final re-assemble le datagramme puisque les fragments peuvent prendre des chemins différents. Les décisions de routage sont basées sur une table de routage qui contient les informations relatives aux différentes destinations possibles et au moyen de les atteindre. Ainsi, modifier les tables revient à modifier les routes empruntées par les datagrammes.

1.2.3 Le protocole ICMP

ICMP (*Internet Control Message Protocol*) [14] est un mécanisme utilisé par les passerelles et les machines pour échanger des messages d'erreur ou de contrôle. Il utilise IP comme le font les protocoles de niveau supérieur, tels que TCP et UDP. Il fait partie intégrante du protocole IP et doit être intégré dans chaque module IP.

Un message ICMP transite dans le champ de données d'un datagramme IP. Chaque message ICMP a sa propre structure mais ils commencent tous par trois champs identiques : un champ type qui détermine la famille de problèmes, un champ code qui précise la nature du problème et un champ vérification du message (*checksum*). Une douzaine de messages sont utilisés pour superviser et contrôler le réseau IP. Le tableau T.3 regroupe tous les types de messages ICMP.

Type de champ	Type de message ICMP
0	Réponse d'écho
3	Destinations non accessibles
4	Ralentir l'émission du message
5	Redirection du message
8	Demande d'écho
11	Problème de temps
12	Problème de paramètre
13	Demande d'estampillage de temps
14	Réponse d'estampillage de temps
15	Demande d'informations
16	Réponse d'informations
17	Demande de masque
18	Réponse de masque

Tableau T.3 - Les différents types de messages ICMP

Les messages **demande d'écho** (*Echo*) et **réponse d'écho** (*Echo Reply*) nous intéressent plus particulièrement. En effet, nous les utilisons lors de l'initialisation d'une communication écran de contrôle - *concentrateur* (chapitre C, §6.3) pour savoir si le *concentrateur* (§4.2.1) est accessible ou non. Le principe est le suivant : une machine A envoie un message *Echo* à une station B ; la machine B répond par un message *Echo Reply* ; dans sa réponse, B inverse les adresses IP d'émetteur et de récepteur, et renvoie les autres champs sans les modifier. Les messages ont un champ **numéro de séquence**, ce qui permet à la station A de savoir quel processus a envoyé un *Echo* quand elle reçoit un *Echo Reply*. Le programme d'application *ping* utilise ce mécanisme pour savoir si une machine est présente (*alive*).

1.2.4 Le protocole TCP

TCP (*Transmission Control Protocol*) [15] complète le protocole IP pour assurer un transport

fiable de l'émetteur au destinataire. Dans le modèle en couches, il se situe au niveau de la couche transport (figure B.3). IP fonctionne de tronçon en tronçon en **mode non connecté** alors que TCP fonctionne de bout en bout en **mode connecté**. TCP permet l'établissement d'un circuit virtuel entre deux programmes d'application distants. Il traite les données venant de la couche supérieure comme une suite d'octets qu'il découpe en segments d'une longueur inférieure à 64 Koctets. Un segment est l'unité de transfert de base de TCP. Il est formé d'un en-tête et d'un corps. L'en-tête est composé notamment des champs :

- **port_source** et **port_destination**
  Ils sont utilisés pour distinguer les différents programmes d'application qui s'exécutent sur une même machine. Le fait de pouvoir distinguer plusieurs destinations sur une machine donnée permet aux programmes d'application d'émettre et de recevoir des données de façon indépendante. Des numéros de ports ont été prédéfinis officiellement [16] pour des applications telles que Telnet (numéro 23) ou FTP (numéros 20 pour le transfert et 21 pour le contrôle). Les applications UNIX utilisent, par convention, les numéros de port 256 à 1024.
- **numéro de séquence**
  Il donne le numéro du premier octet de données dans le segment, par rapport au début de connexion. Les octets peuvent ainsi être réordonnés à l'arrivée et la duplication de segments transmis est également évitée.
- **numéro d'acquittement**
  Il indique le numéro du prochain octet que l'on veut recevoir et constitue donc un acquittement de tous les octets reçus antérieurement.

Le protocole TCP contrôle les numéros de séquence des segments transmis. L'émetteur arme une temporisation lors de l'envoi d'un segment et attend un acquittement du récepteur. Si cet acquittement n'est pas reçu à l'expiration de la temporisation, le segment est considéré perdu et il est retransmis.

TCP possède un mécanisme de contrôle de flux de bout en bout. Il permet au récepteur de limiter la transmission jusqu'à ce qu'il ait suffisamment de mémoire libre pour recevoir des octets supplémentaires. Dans les situations extrêmes, la transmission d'octets peut être arrêtée pour éviter des situations de blocage. Les machines dont les vitesses varient dans une large gamme peuvent ainsi communiquer efficacement.

Le protocole TCP offre une interface qui permet l'établissement et la libération d'une connexion entre programmes d'application, ainsi que le transfert de données sur cette connexion. Un exemple est l'interface des *sockets* en mode TCP (§1.3.1). Celle-ci est composée de primitive, utilisées par les programmes d'application tels que Telnet ou FTP et qui permettent :

- d'établir une connexion ;
- d'accepter l'établissement d'une connexion ;
- d'envoyer des données sur une connexion établie ;
- de réceptionner des données ;
- de libérer une connexion ;
- d'arrêter brutalement une connexion.

Une analogie avec TCP est l'appel téléphonique : il faut préalablement établir une connexion entre deux interlocuteurs pour qu'il y ait dialogue.

Dans le projet Vivitron nous utiliserons ce protocole de contrôle de transmission pour exécuter les commandes. Sa fiabilité permet d'assurer le transport de la commande depuis les écrans de contrôle jusqu'aux *concentrateurs* (§4.2.1). Le seul problème qui peut se poser se trouve lors de la connexion

écran de contrôle - *concentrateur*. Si le *concentrateur* ne répond pas, il faut attendre l'expiration du *timeout* (60 secondes) pour refaire une nouvelle commande. Ce problème a été résolu en utilisant le protocole ICMP (un *ping* avec un *timeout* réduit à quelques millisecondes) avant chaque connexion TCP.

1.2.5 Le protocole UDP

Dans le modèle en couches, UDP (*User Datagram Protocol*) [17] se situe au même niveau que TCP mais ne peut assurer qu'un transport non fiabilisé en **mode non connecté.** Les messages UDP sont transmis sans établissement de connexion préalable, sans garantie du bon acheminement ni du respect de la séquence des données transmises, et sans contrôle de flux. Ils peuvent donc être perdus, dupliqués, déséquencés ou retardés. Ce sont les programmes d'application qui doivent gérer ces problèmes. UDP utilise également les numéros de **port_source** et **port_destination** mais en les distinguant de ceux de TCP.

Le protocole UDP est en fait utilisé dans les cas où la perte de données est sans grande conséquence. Les messages UDP peuvent être envoyés en grande quantité sans être bloqués par une connexion. UDP est utilisé pour la diffusion, dans des programmes d'application tels que *rwho* ou *routed*, ou dans des réseaux à faible taux d'erreur avec des applications telles que TFTP (*Trivial File Transfert Protocol).* Dans notre projet Vivitron, UDP est utilisé pour le transport des mesures. En effet, les mesures sont effectuées régulièrement à une cadence assez rapide (quelques milliers par seconde). La perte d'une mesure est donc sans conséquence et l'absence d'une **connexion bloquante** permet d'envoyer les mesures sans ce soucier de la présence du destinataire.

Une analogie avec UDP est le courrier postal : les lettres peuvent arriver dans un ordre différent de celui de leur expédition et rien ne garantit que la lettre arrivera à destination.

1.3 Le modèle client - serveur

Un **serveur** est un programme d'application tournant en tâche de fond (démon) qui contrôle une ou plusieurs ressources et qui attend, puis traite, les requêtes des clients. Un **client** est un programme utilisateur qui communique avec un serveur pour obtenir un service donné. Il prend l'initiative de la communication. L'application client réside toujours dans la machine locale, tandis que le serveur peut s'exécuter sur une machine à distance (inversion pour le modèle X-Window [18]).

1.3.1 Les *sockets*

Les **sockets** qualifient une interface entre les programmes d'application et les protocoles TCP/IP, introduits par UNIX à partir de la version 4.2 BSD. Cette interface permet à la fois la communication entre processus locaux et entre processus distants. Un **socket** est une extrémité de connexion dans une communication client - serveur. Il est constitué par la combinaison de l'adresse IP, du numéro de port (identifiant localement le programme d'application connecté) et du protocole de

communication (UDP ou TCP). Il se comporte de façon identique à un fichier : sa création par une primitive particulière *socket()* retourne un descripteur qui peut être utilisé dans les opérations classiques (*read*, *write*, *close*, *send*, etc.). Mais un descripteur de *socket* n'est pas un descripteur de fichier. En effet, il pointe sur une structure de données contenant les caractéristiques du *socket*.

Les *sockets* facilitent la mise en place des applications accédant aux protocoles de communication, mais ils ne permettent que l'échange de données non structurées, ce qui est leur principal défaut. Pour pallier cette lacune, des mécanismes de plus haut niveau comme les RPC, ont été mis en place.

1.3.2 Les RPC

SUN Microsystems a développé un système de **fichiers distribués** (NFS) qui permet l'accès à des fichiers partagés (locaux ou distants) de façon transparente pour l'utilisateur. Le protocole NFS (*Network File System*) [19] s'appuie sur les **appels de procédures distantes** (RPC) et sur le mécanisme de **représentation externe** (XDR). Ces deux derniers modules ont été conçus indépendamment de NFS pour permettre leur utilisation dans d'autres applications. Par rapport au modèle OSI, les couches session et présentation sont représentées respectivement par RPC et XDR, tandis que NFS correspond à la dernière couche application (figure B.3).

Les procédures RPC (*Remote Procedure Call*) [20] ont été rendues publiques par SUN Microsystems. Ce protocole de communication permet à un client de demander à un serveur distant d'exécuter une procédure déterminée avec retour des résultats. Il est basé sur l'interface des *sockets* qui devient complètement transparente pour le programmeur et offre une librairie de fonctions qui permet de développer des applications personnalisées.

L'utilisation de XDR (*External Data Representation*) [20] autorise l'échange de variables structurées entre machines ayant des architectures matérielles et des systèmes d'exploitation différents. XDR définit une représentation des variables indépendante de toute machine. Il définit la taille, l'ordre des octets, le type *chaîne de caractères*, le type *tableau*, etc. La machine émettrice utilise XDR pour passer de la représentation locale à la représentation externe XDR. Une fois les données transférées sur la machine réceptrice distante, celle-ci fait appel à XDR pour effectuer la conversion de la représentation externe XDR à sa représentation locale. La librairie XDR, sous-ensemble de la librairie RPC, fournit les routines pour les différents types de base mais le programmeur à la possibilité d'écrire ses propres routines afin d'introduire des structures de données particulières. C'est cet aspect qui va être utilisé dans le projet Vivitron pour introduire des structures de données configurables dynamiquement (leur taille varie automatiquement).

Un serveur RPC est défini de façon unique par son numéro de programme, sa version et le protocole utilisé (TCP ou UDP). Il peut être amené à gérer plusieurs procédures, distinguées par un numéro d'identification. Le client RPC doit être capable de s'adresser directement au serveur RPC mais, pour cela, il a besoin du numéro de port du serveur. C'est le service *port mapper* (*portmap*) qui va lui fournir cette information si le serveur RPC désigné à été déclaré auparavant au niveau de ce service.

Les programmes client et serveur RPC peuvent être mis en œuvre facilement par l'outil *rpcgen*. La génération d'une application RPC à l'aide de *rpcgen* peut se décomposer en 4 phases :

- Description par le développeur, en langage RPC, des structures de données utilisées par la procédures RPC (fichier *appli.x*).

- Pré-compilation par rpcgen du fichier *appli.x* et création des squelettes C du client et du serveur (*appli_clnt.c et appli_svc.c*), d'un fichier de données au format XDR (*appli_xdr.c*) et d'un fichier en-tête contenant la déclaration des structures de données (*appli.h*).
- Ecriture par le développeur du code de mise en œuvre de la procédure RPC (proc.c) et de l'appel à cette procédure (client.c), et association de ce code avec les squelettes.
- Compilation par le développeur du code source résultant et génération des fichiers exécutables *serveur* et *client*.

L'inconvénient de cet outil est que l'on n'a aucun moyen d'agir sur les modèles du squelette du serveur et du client ni sur le fichier de données au format XDR générés par *rpcgen*. Dès qu'on va relancer *rpcgen*, ces modules vont être écrasés et toute modification apportée manuellement par le programmeur va disparaître. La solution que nous avons apportée pour le projet Vivitron est de bien définir les applications RPC client/serveur, de les créer une première fois à l'aide de *rpcgen*, pour les personnaliser, les transposer à d'autres structures de données et les modifier par la suite, sans utiliser *rpcgen*.

1.3.3 X-Window

Un système de fenêtrage comprend toutes les fonctionnalités permettant à une application de créer une ou plusieurs fenêtres à l'écran et d'utiliser ces fenêtres pour dialoguer avec l'utilisateur. X-Window (appelé aussi X) est un système de fenêtrage développé au MIT (*Massachussets Institute of Technology*) en collaboration avec DEC (*Digital Equipment Corporation*). Il est fourni avec des librairies de fonctions généralement écrites en C et composées d'une interface de programmation bas niveau, appelée *Xlib*, permettant d'exploiter le protocole X, et d'un module haut niveau appelé *Xt* ou *toolkits Intrinsics* permettant de créer et d'utiliser des objets de dialogue appelés *widgets* (menus, boutons, boîtes de dialogue, ascenseurs, etc.) (figure B.5). X-Window n'est pas une interface graphique à proprement parler : il fournit les éléments de base pour la conception de l'interface graphique mais il ne définit pas son comportement. Les interfaces graphiques telles que Motif d'OSF ou OpenWindows de SUN, utilisent un niveau de programmation plus élevé, basé sur la librairie *Xt* ou d'autres librairies. Toutes nos applications utilisateur (SL-GMS [27] et O_2 [26]) ont une interface graphique X-Window.

Le système X-Window a été conçu de manière à être le plus indépendant possible du matériel, du système d'exploitation et des moyens de communication. Il est basé sur un modèle client/serveur, mais avec une inversion par rapport au modèle classique : le serveur X se trouve dans la machine locale tandis que le client X peut s'exécuter sur une machine distante. X permet, de manière distribuée, l'exécution d'applications graphiques à travers un réseau. Il n'est pas nécessaire que l'interface utilisateur se trouve sur la même machine que l'application : une application s'exécutant sur une machine peut s'afficher soit sur le même écran que la machine, soit sur un écran à distance. Les applications développées avec le système X-Window sont indépendantes du matériel d'affichage : elles n'accèdent pas directement au matériel mais passent par un intermédiaire, le serveur X qui, lui, est dépendant du matériel d'affichage et d'entrée.

Le serveur X est un programme d'application qui gère le **display**, c'est-à-dire les outils d'affichage et de dialogue avec l'utilisateur : souris, clavier, écran, etc. Un *display* est constitué d'un ou plusieurs écrans graphiques, d'un clavier et d'une souris. Par *display*, il ne peut y avoir qu'un seul serveur, lequel peut traiter les requêtes d'un ou plusieurs clients X.

Le client X représente l'application utilisateur. Pour écrire sur l'écran, lire le clavier ou la souris, il envoie des requêtes d'affichage ou de dialogue au serveur X. Il peut se connecter à un ou plusieurs serveurs X.

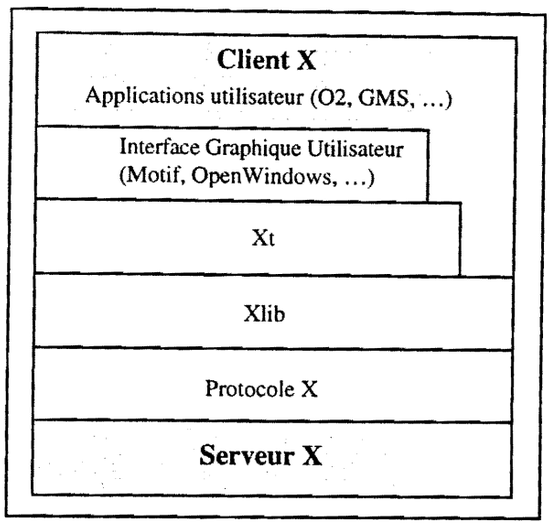


Figure B.5 - Structure logicielle de X-Window

Le serveur et le client sont deux programmes d'application distincts qui peuvent s'exécuter sur la même machine ou sur des machines distantes, communiquant entre eux par l'intermédiaire du protocole X qui utilise les *sockets* (§1.3.3.2) [18].

1.3.3.1 La connexion au serveur X

Le client X doit établir une connexion avec un serveur X avant de formuler une requête X quelconque. Cette connexion est effectuée de façon transparente par la fonction *XOpenDisplay()* de la Xlib à laquelle on indique les références du serveur X à contacter. Ces références sont spécifiées avec la syntaxe suivante : **nom_machine:numéro_serveur.numéro_écran**

- **nom_machine** est le *hostname* de la machine ou son adresse IP. Ce champ est séparé du suivant par ":" si le protocole de communication utilisé est TCP/IP ou "::" s'il s'agit des protocoles DECnet (§1.3.3.3).
- **numéro_serveur** indique le serveur concerné car il se peut que plusieurs *displays* soient attachés à cette machine. Si les connexions s'effectuent sous le protocole TCP, les *displays* sont numérotés à partir de 0, et le numéro de port utilisé pour la connexion est *6000 + n* où *n* est le numéro du *display*.
- **numéro_écran** indique quel écran doit être utilisé quand le serveur gère plusieurs écrans. Ce champ est optionnel, mais nous allons l'utiliser dans le projet Vivitron.

Par la suite, tous les messages échangés par le client et le serveur utiliseront cette connexion qui sera identifiée par une structure de données de type ***Display***.

1.3.3.2 Les fenêtres

L'objet graphique de base utilisé par le serveur X est la fenêtre. Une fenêtre est une zone délimitée de l'écran accessible par l'intermédiaire d'un identificateur unique. C'est la représentation du

client X pour le serveur X. Le client X est obligé de créer une fenêtre pour pouvoir réaliser des opérations graphiques ou dialoguer avec d'autres programmes d'application. La création d'une fenêtre par la fonction *XCreateWindow()* ne provoque pas sa visualisation à l'écran ; la visualisation n'est pas nécessaire si le programme d'application n'effectue que des opérations de dialogue.

Le serveur X organise les différentes fenêtres en ordre hiérarchique. Lors de son démarrage, il crée une fenêtre pour chaque écran qu'il gère : c'est la **fenêtre racine** (*root window*). Celle-ci occupe l'écran dans sa totalité et aucun client X ne peut la modifier. Elle n'a pas de fenêtres "mères" mais elle est la "mère" des premières fenêtres des différents clients X. Une fenêtre est toujours la "fille" d'une autre fenêtre et peut également avoir des fenêtres sœurs (figure B.6). Sa dimension peut dépasser celle de sa fenêtre "mère" mais sa visibilité est limitée à la visibilité de sa fenêtre "mère".

La gestion des fenêtres présentes à l'écran (déplacement, empilement, etc.) est effectuée par un client X particulier, le *window manager*. Ce client tourne sur la machine où se trouve le serveur. Il n'a pas accès au contenu des fenêtres, mais prend en charge leur encadrement ou leur icône et peut mettre fin aux programmes ou exécuter d'autres clients X. Il permet à l'utilisateur de gérer un ensemble hétéroclite de fenêtres à l'écran.

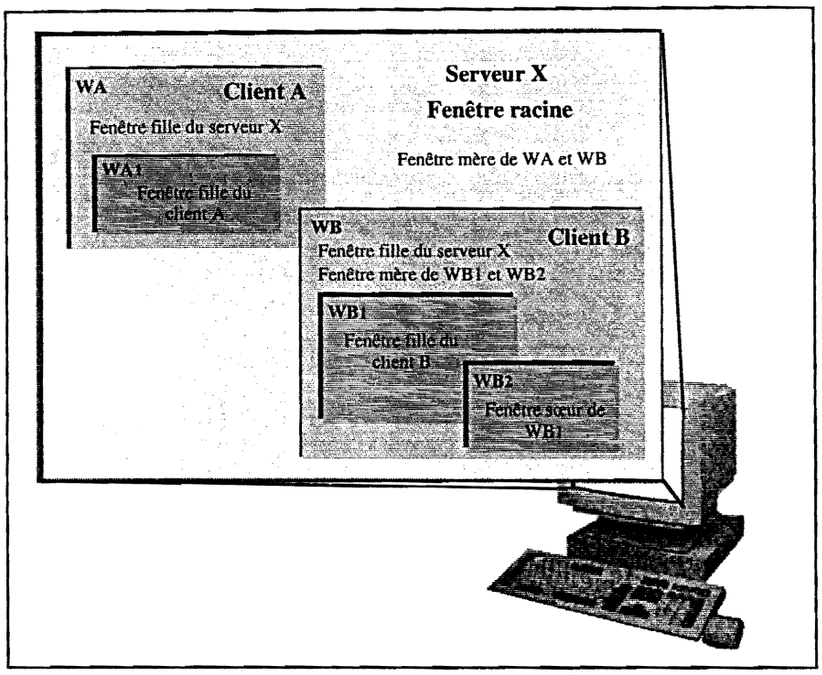


Figure B.6 - Organisation hiérarchique des fenêtres

1.3.3.3 Le protocole X

Le protocole X est totalement indépendant d'un quelconque type de réseau. Lorsque le client et le serveur résident dans la même machine, le protocole X utilise des moyens de communication inter-processus tels que la **mémoire partagée**. S'ils se trouvent dans des machines différentes, le protocole X utilise comme moyen de communication, la plupart du temps, les protocoles DECnet ou TCP/IP, également indépendants du type de réseau. Ces protocoles sont transparents pour l'application.

Pour des raisons de performance de communication, le protocole X est de type **asynchrone**, ce qui permet à un client qui envoie une requête de ne pas attendre de confirmation du serveur ; il en est de même pour le serveur. Au niveau du client, les requêtes sont temporairement sauvegardées dans leur ordre d'arrivée, dans des zones tampons et envoyées régulièrement selon un fonctionnement bien précis (figure B.7). L'avantage de cette méthode est que les messages sont envoyés par blocs et donc, le nombre d'accès au réseau est considérablement réduit. L'envoi du tampon est effectué par le client lorsque :

- Le tampon est rempli.
- Le client X est en attente d'un événement qui tarde à se manifester. L'événement attendu peut-être lié à une précédente requête ; il faut donc transmettre les requêtes suivantes au serveur (*XNextEvent()*).
- Le client X émet une requête nécessitant une réponse.
- Le client X provoque de manière explicite la transmission de la zone tampon (*XFlush()*, *XSync()*). Notons que *XFlush()* se limite à envoyer la zone tampon tandis que *XSync()* transmet la zone tampon et bloque le client X le temps que toutes les requêtes soient traitées par le serveur X. La transmission des événements X avec *XSync()* est une transmission en mode synchronisation. Elle sera utilisée dans la nouvelle version du système informatique du Vivitron.

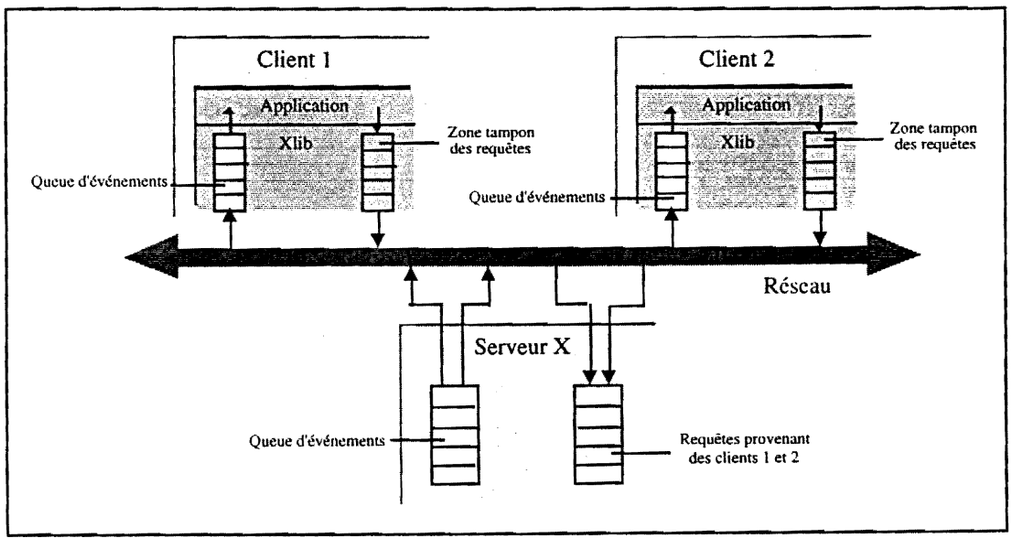


Figure B.7 - Transmission asynchrone des requêtes et des événements

Au niveau du serveur, le mécanisme des zones tampons n'est pas utilisé fréquemment car la nature des messages à transmettre au client (événements, réponses et erreurs) leur confère un caractère d'urgence.

1.3.3.4 Les événements

Les événements sont le moyen dont dispose le serveur X pour informer le client X de toute action provenant du clavier, de la souris ou d'un autre client X. Ils ne sont pas envoyés dans le désordre à n'importe quel client X. Le client qui veut recevoir un événement doit d'abord le sélectionner par rapport à une fenêtre (requête *XSelectInput()*). A chaque type d'événement est associé

un ou plusieurs masques, par exemple *ButtonPressMask* pour la souris ou *KeyPressMask* pour le clavier. Le serveur X n'enverra au client X que les événements correspondant au(x) masque(s) affecté(s) à la fenêtre choisie par le client. Il existe cependant des événements auxquels ne sont pas associés de masques car ces événements sont envoyés d'office aux clients X et ne peuvent donc pas être sélectionnés. C'est le cas, par exemple, de l'événement de type ***ClientMessage*** permettant l'envoi de messages entre clients X.

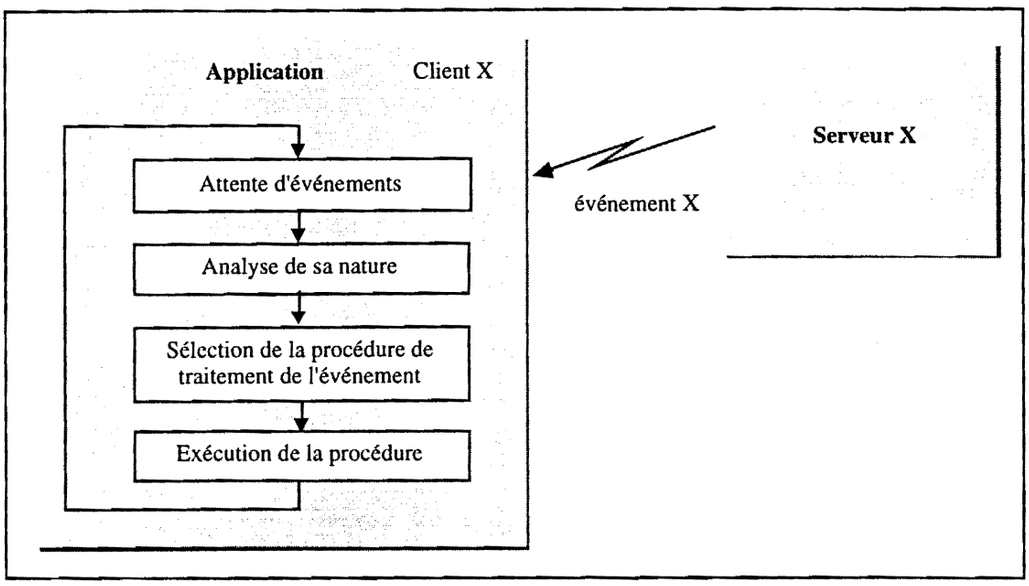


Figure B.8 - Boucle de traitement des événements X

Dans une gestion par événements, l'application est construite autour d'une boucle principale qui scrute les événements envoyés par le serveur X (figure B.8). Lorsqu'un événement survient, cette boucle analyse le type d'événement et déclenche une procédure particulière qui va traiter cet événement. Xlib fournit la fonction *XNextEvent()* qui va consulter la queue des événements du client X. Si un ou plusieurs événements sont présents, elle retire de la queue l'événement le plus ancien pour qu'il soit traité. Si aucun événement n'est présent, elle provoque l'envoi des requêtes qui se trouvent dans la zone tampon du client X, si elles existent, et attend un nouvel événement en retour, ce qui bloque l'exécution du client.

Les événements sont identifiés par des structures de données de type ***XEvent***. ***XEvent*** est une structure de la Xlib dont le premier champ, "type", contient la nature de l'événement et dont les autres champs mémorisent les données techniques de l'événement. Par exemple, lorsque l'événement est de type ***ClientMessage***, ***XEvent*** contient un champ *xcl*ient qui est de type ***XClientMessageEvent*** et qui va mémoriser les données concernant cet événement. Il existe une structure par type d'événements. Ceux-ci peuvent être regroupés en quatre groupes différents :

- événements liés aux périphériques d'Entrée/Sortie (souris, clavier, etc.) ;
- événements liés à la gestion des fenêtres ;
- événements liés à la signalisation d'un changement de configuration ;
- événements liés à la communication entre les clients X (*ClientMessage*).

Le groupe qui nous intéresse tout particulièrement est le dernier, car ses événements sont utilisés pour échanger des informations entre clients X. Bien que le protocole X n'ait pas été créé pour le transport

de données, nous allons insérer ce protocole dans la partie communication de notre application Vivitron (voir chapitre B, §7).

La librairie Xlib met à la disposition du client X une fonction, *XSendEvent()*, lui permettant d'envoyer un événement vers un autre client X connecté au même serveur X. N'importe quel type d'événement peut être envoyé, mais le plus approprié pour la communication entre les clients est le type ***ClientMessage***. En effet, ce type d'événement est envoyé d'office aux clients X car il ne peut pas être sélectionné, faute de masque. Sa structure de données, ***XClientMessageEvent*** est définie ainsi :

```
typedef struct {
        int         type ;          /* type de l'événement                          */
        u_long      serial ;        /* No de la dernière requête traitée par le serveur */
        BOOL        send_event ;    /* origine de l'envoi : serveur ou client       */
        Display     *display ;      /* identifie la connexion d'où provient l'événement */
        Window      window ;        /* fenêtre qui a sélectionné l'événement        */
        Atom        message_type ;  /* type de message                              */
        int         format ;        /* format du message : 8 bits, 16 bits ou 32 bits */
        union {
            char    b[20];          /*      utilisés suivant la valeur de format    */
            short   s[10];          /*                                              */
            long    l[5];           /*                                              */
        } data ;
} XClientMessageEvent ;
```

Dans notre projet Vivitron nous ferons communiquer les écrans graphiques et les *concentrateurs* (§4.2.1) par des événements X (chapitre C, §6) de type ***ClientMessage*** avec des messages composés de 20 caractères (format 8 bits).

Pour clore ce paragraphe, nous ferons un bref commentaire sur les deux gestionnaires d'erreurs fournis par la Xlib. Le premier, *ErrorHandler*, assure la gestion des événements d'erreur envoyés par le serveur X. Le deuxième, *IOErrorHandler*, traite les erreurs d'entrée/sortie (coupure réseau, accès au serveur X interrompu, tentative d'accès à un serveur X inexistant, etc.). Les fonctions de ces deux gestionnaires peuvent être remplacées par des fonctions utilisateur. Cette permutation peut être réalisée à l'aide des fonctions *XSetErrorHandler()* et *XSetIOHandler()* appartenant à la Xlib. Ces gestionnaires nous seront très utiles pour résoudre les problèmes dus aux erreurs d'entrée/sortie (chapitre C, §6.4.1).

2 Le bus VME

2.1 Introduction

Pour augmenter la rapidité de traitement d'un système informatique, l'un des moyens est d'augmenter le nombre de processeurs. Dans les systèmes de ce type (systèmes multiprocesseurs) les différentes tâches sont exécutées par plusieurs processeurs. Il est alors possible d'utiliser ces derniers au-dessous de leurs possibilités maximales ce qui permet une re-configuration du système en cas de panne. Cette structure multiprocesseur autorise le parallélisme : les processeurs se partagent soit une tâche donnée soit un ensemble de tâches. Dans un tel système la défaillance d'un processeur a pour seule conséquence le ralentissement de la rapidité d'exécution des tâches. La fiabilité et la modularité sont ses principaux avantages.

Dans un environnement multiprocesseur chaque processeur dispose de ses propres bus et de sa propre mémoire mais ils peuvent partager une mémoire commune et les périphériques. L'échange des informations entre processeurs se fait par un bus commun à tous les processeurs : le **bus système**. Chaque processeur travaille de manière asynchrone, les transferts de données entre modules ne sont pas cadencés par un signal de synchronisation (type horloge) mais par des signaux de dialogue (demande et acquittement) qui précèdent ou suivent ces transferts. Afin de prévenir les conflits, dans le cas de plusieurs demandes de bus simultanées, un arbitrage des priorités est nécessaire. Un bus système est constitué essentiellement :

- d'un bus de transfert de données ;
- d'un bus d'adresse ;
- d'un bus d'arbitrage ;
- d'un bus de gestion des interruptions.

Le bus VME (*Versa Module Eurocard*) est un bus multiprocesseur standardisé et largement diffusé dans le monde industriel. Le concept VME a commencé sa carrière par le simple bus VME constitué d'un seul connecteur de 96 contacts, appelé P1, comportant 16 bits de donnée et 24 bits d'adresse. Cette première version fût présentée à Munich en octobre 1981 par les constructeurs Motorola, Mostek et Philips sous le nom de bus VME Révision A. Depuis 1982/83, son architecture s'est étendue grâce au second connecteur de 96 contacts, appelé P2, Ce dernier présente sur la rangée du milieu (rangée b) l'extension du bus VME à 32 bits de donnée et 32 bits d'adresse et sur les rangées externes (rangées a et c) les connections d'Entrée/Sortie ou les connections pour le bus local VSB (32 bits de donnée et d'adresse multiplexés). Cette version est connue sous le nom de Révision B, standardisée en 1982 par le CEI (*Comité Electrotechnique International*) sous le nom CEI-821 et en 1983 par l'IEEE sous le nom IEEE-P1014. La Révision C a fait l'objet d'une norme par l'organisme IEEE (*Institute of Electrical and Electronics Engineers*) en 1987 : la norme IEEE 1014-87. Actuellement, la révision D du bus VME est soumise à approbation, elle permet entre autres le transfert d'adresses et de données de 64 bits.

2.2 Caractéristiques

Le bus VME est un bus asynchrone et non multiplexé (la révision D utilise le mode multiplexé pour effectuer des transferts de 64 bits). Sa capacité d'adressage est de 4 Goctets et sa vitesse maximale de transfert de données de 40 Moctets/s. Il peut accepter un nombre maximum de 21 cartes par châssis VME. Les interruptions sont gérées sur 7 niveaux de priorités par un système centralisé (un seul contrôleur) ou par un système distribué (plusieurs contrôleurs d'interruptions dans le même châssis VME).

Ce bus accepte une architecture modulaire, il est basé sur le concept de maître/esclave. Un **maître** est un module actif (processeur, contrôleur d'interruption, etc.) qui transfère des données à destination ou en provenance d'un module passif appelé **esclave** (carte mémoire, carte d'Entrée/Sortie, etc.). Les échanges de données peuvent être effectués sous 3 formats différents 8, 16 ou 32 bits permettant de partager des ressources communes (mémoires, Entrées/Sorties) et des périphériques ayant des caractéristiques de vitesse différentes. L'accès au bus de transfert de données est géré par un élément commun, appelé **arbitre**, et conditionné par un niveau de priorité (4 niveaux). L'arbitre répond à une requête (*Bus Request*) par un acquittement (*Bus Grant)* afin d'éviter des éventuelles collisions lors des échanges.

Le bus local VSB permet d'alléger le transfert de données sur le bus principal (le bus VME). Il est notamment utilisé par les processeurs pour accéder à leur mémoire privée (carte mémoire) ou à des cartes d'Entrée/Sortie sans utiliser le bus VME. Le bus VSB est un bus asynchrone, multiprocesseur et multiplexé, avec une capacité d'adressage de 4 Goctets et une vitesse maximale de transfert de données de 80 Moctets/s. Il peut accepter un nombre maximum de 6 cartes par bus VSB et un ou plusieurs bus VSB peuvent être installés dans un châssis VME. Les interruptions sont gérées par vectorisation ou par scrutation et les accès au bus par arbitrage sériel ou parallèle.

Un bus série, appelé bus VMS, permet les liaisons série interprocesseur. Le VMS est un bus série synchrone utilisé pour le transfert de messages entre des cartes du même châssis ou de châssis différents. Il possède une ligne de données et une horloge cadencée à une vitesse maximale de 2.9 Mbits/s.

2.3 Le transfert de données

Bus d'adresse	A01-A31	24 bits sur P1 et 8 bits sur P2
Bus de donnée	D00-D31	16 bits sur P1 et 16 bits sur P2
Contrôle d'adresse	AS* AM0-AM5	Validation d'adresse Modifications d'adresses
Contrôle de donnée	DS0*-DS1* WRITE* LWORD* DTACK* BERR*	Validation de donnée (Lecture/Ecriture) Mot long Donnée disponible Erreur bus

Tableau T.4 - Les signaux du Bus de Transfert de Données

Toutes les données sont transférées sur le **Bus de Transfert de Données** (DTB). Le DTB est utilisé par les maîtres pour transférer des données vers ou depuis les esclaves. Le tableau T.4 nous montre les lignes d'adresse, de donnée et de contrôle qui composent ce sous-bus. Ce dernier supporte 5 cycles de transfert de données :

- cycle de lecture ;
- cycle d'écriture ;
- cycle indivisible de lecture-modification-écriture ;
- cycle uniquement d'adressage sans transfert de données ;
- cycle à accès séquentiel (par blocs de 256 octets maximum).

2.3.1 L'adressage

Le bus VME avec 24 bits permet d'adresser 16 Moctets et avec l'extension sur le connecteur P2, les 32 bis permettent le décodage de 4 Goctets. L'adressage des esclaves s'effectue à chaque cycle de lecture et à chaque cycle d'écriture. Un décodage d'adresse complexe pour les esclaves serait nécessaire, mais la norme VME définie 3 espaces adressables :

- l'**adressage court** qui autorise un adressage de 64 Koctets ;
- l'**adressage standard** qui autorise un adressage de 16 Moctets ;
- l'**adressage étendue** qui autorise un adressage de 4 Goctets.

Ces types d'adressage sont définis par la combinaison des 6 lignes de Modification d'Adresse (*Address Modifier*). Quand un esclave reçoit un adressage court, d'après le code des lignes AM, il ignore les adresses A16-A31. Pour un adressage standard, les lignes A24-A31 sont ignorées. La norme VME spécifie que l'esclave doit répondre au moins à un des trois modes d'adressage.

Notons que la ligne d'adresse A00 n'existe pas. L'adresse 0 est différenciée à partir des deux signaux de Validation de Donnée (*Data Strobe*) DS0* qui sélectionne l'octet impair (octet faible) et DS1* qui sélectionne l'octet pair (octet fort).

Le transfert de données s'effectue sur un octet, sur un mot de 16 bits ou sur un mot long de 32 bits signalé par LWORD* (*Long WORD*) :

- L'octet est transféré sur les lignes de données D00-D07 et validé par DS0*.
- Le mot de 16 bits est transféré sur les lignes D00-D07 (octet faible) et D08-D15 (octet fort) et les signaux DS0* et DS1* permettent la sélection respective de l'octet impair et de l'octet pair. Il est aligné dans l'espace mémoire ce qui signifie que son adresse est paire et que la ligne d'adresse A01 n'est donc pas utilisée.
- Le mot long de 32 bits est transféré en un seul cycle VME par la validation du signal LWORD*. Son adresse est alignée sur le mot pair (A01 = 0) et les données sont transférées sur les lignes D00-D31.

2.3.2 Le code Modificateur d'Adresse

Ce code, composé de 6 lignes de modification d'adresse AM0-AM5, permet au maître de choisir la largeur du bus d'adresse. Le tableau T.5 nous montre les différents codes utilisés par la norme VME [21].

Code AM (Hexa)	Nbr de bits d'adresse	Type de transfert
0x3F	24	Accès ascendant en mode superviseur standard
0x3E	24	Accès au programme en mode superviseur standard
0x3D	24	Accès aux données en mode superviseur standard
0x3B	24	Accès ascendant en mode utilisateur standard
0x3A	24	Accès au programme en mode utilisateur standard
0x39	24	Accès aux données en mode utilisateur standard
0x2D	16	Accès Entrée/Sortie en mode superviseur court
0x29	16	Accès Entrée/Sortie en mode utilisateur court
0x10-0x1F	-	Définis par l'utilisateur
0x0F	32	Accès ascendant en mode superviseur étendu
0x0E	32	Accès au programme en mode superviseur étendu
0x0D	32	Accès aux données en mode superviseur étendu
0x0B	32	Accès ascendant en mode utilisateur étendu
0x0A	32	Accès au programme en mode utilisateur étendu
0x09	32	Accès aux données en mode utilisateur étendu
Note : *Accès ascendant* signifie accès séquentiel de la mémoire (utilisé dans le cycle à accès séquentiel).		

Tableau T.5 - Les codes modificateurs d'adresse utilisés par la norme VME

Le modificateur d'adresse permet aussi au maître d'envoyer des informations supplémentaires lors d'un transfert de données comme le type de cycle bus. Ces informations vont spécifier diverses fonctions.

- **Configuration dynamique du système**

  L'utilisation du modificateur d'adresse est obligatoire (règle 2.9 [21]). Dans l'éventualité de configurer un système dans ce mode de fonctionnement, un esclave peut être alloué à plusieurs maîtres par un code de modificateur d'adresse différent. Une telle précaution évite une panne générale en cas d'un mauvais fonctionnement du maître, l'esclave restant alors accessible par les autres maîtres. Chaque maître peut donc sélectionner un esclave dynamiquement à l'intérieur de l'espace adressable VME ce qui permet un découpage en partitions de cet espace adressable.

- **Accès privilégié**

  La sélection d'une carte esclave se fait en deux étapes : le décodage des adresses A01-A31, puis le décodage des lignes AM0-AM5. Un esclave répond à différentes adresses, dépendantes du code modificateur d'adresse. De ce fait, dans un environnement multiprocesseur, l'esclave peut avoir un accès privilégié en **mode superviseur** (appels systèmes autorisés). Chaque maître lors de l'accès donne son niveau de privilège à partir des codes modificateurs d'adresse (accès en **mode superviseur** ou en **mode utilisateur**).

- **Accès à une séquence de données**

  L'accès à une séquence de données est le cycle à accès séquentiel BLT (*BLock Transfer*). Il peut être utilisé par le modificateur d'adresse pour transférer des données à grande vitesse. Pendant le cycle, le maître ne génère l'adresse qu'une seule fois au lieu de générer une adresse à chaque

transfert de données. Il commence le cycle de transfert d'une manière classique, comme un cycle de lecture ou d'écriture. Il présente à l'esclave à la fois une adresse et un code modificateur d'adresse à accès séquentiel. L'esclave stockera l'adresse présente sur le bus d'adresse dans un registre spécialisé (un compteur). Le maître effectue le premier transfert et ne désactive pas le signal AS* (*Address Strobe*). Il laisse actif ce signal pendant toute la séquence de transfert. Après la prise en compte par l'esclave de la donnée, ce dernier émet un DTACK* (*Data Transfer ACKnowledge*) et le maître génère à nouveau les signaux DS0* et DS1* afin de continuer l'opération. L'esclave incrémente l'adresse stockée dans le compteur, ce qui décharge le maître du calcul et du changement de l'adresse à chaque transfert de données. L'accès à une séquence de données ne peut pas être interrompu par une demande d'accès au bus généré par un autre maître car cette demande ne peut être prise en compte que lorsque le signal AS* est désactivé.

- **Sélection d'un système de gestion de mémoire**

  La séparation des espaces d'adressage et la partition de ces espaces en segments (blocs-mémoire) est prise en charge par un système de gestion de mémoire. Ce système traduit l'adresse logique en adresse physique. L'adresse logique, fournie par le processeur, est généralement constituée de deux parties : le numéro de segment et le déplacement à l'intérieur du segment. L'adresse physique sera calculée en ajoutant le déplacement à l'adresse physique de base qui est fournie soit par l'utilisateur soit par le système d'exploitation. L'ensemble des informations relatives à un segment est contenu dans un registre appelé **descripteur**.

  A chaque tâche est alloué un certain nombre de segments. Lorsque la **tâche superviseur** exécute plusieurs **tâches utilisateur** elle doit modifier le contenu des descripteurs du système de gestion de mémoire ou commuter d'un système de gestion à un autre. Ce dernier cas, plus performant, se voit alloué un système de gestion mémoire à chaque tâche. La sélection d'un système de gestion de mémoire parmi plusieurs peut se faire par le modificateur d'adresse.

2.4 Le fonctionnement en mode multiprocesseur

Dans un environnement multiprocesseur, les maîtres sont autorisés à partager le bus de transfert de données. L'accès au bus est géré par un arbitre qui aura pour rôle de recevoir les demandes, de gérer les priorités et d'attribuer le bus. Il y a deux types d'arbitrage : l'**arbitrage local** et l'**arbitrage central**. Dans l'arbitrage local, il y a un arbitre par demandeur de bus (c'est le cas du bus VSB) et l'arbitrage est réalisé par le maître qui est en possession du bus. Dans l'arbitrage central, un seul arbitre est présent dans le système. Il reçoit toutes les requêtes et attribue le bus au demandeur le plus prioritaire suivant une technique de gestion des priorités (priorité fixe, tournante, etc.).

C'est l'arbitrage central qui est utilisé par le bus VME. Lorsqu'un maître A est en possession du bus et qu'un maître B, de priorité plus élevée, émet une demande d'accès bus, l'arbitre enregistre sa demande et puis active le signal de libération de bus BCLR* (*Bus CLeaR*). Ce signal informera le maître A qu'un autre maître plus prioritaire est en attente. Le maître A libérera le bus à la fin du cycle en cours et l'arbitre pourra donc désactiver le signal BCLR* et accorder le bus au second maître. Le maître A émettra à nouveau sa demande dans l'attente d'être accordée. Les maîtres sont chaînés en guirlande (*daisy chain*) et chacun dispose de 4 niveaux de priorité de demande de bus, assignés aux signaux BR0*-BR3* (*Bus Request*).

Trois techniques d'arbitrage peuvent être utilisées :

- **Priorité fixe** PRI (*PRIority*)
  L'arbitre assigne une priorité fixe à chaque ligne BR, la ligne BR3* ayant le niveau le plus prioritaire. La demande la plus prioritaire sera toujours accordée par l'arbitre. Si un maître de priorité plus faible est en possession du bus, l'arbitre signalera au maître occupant qu'un maître plus prioritaire est demandeur en activant le signal de libération de bus BCLR* (*Bus CLeaR*).
- **Priorité tournante** RRS (*Round Robin Select*)
  Le niveau de priorité n'est plus fixe, le maître qui a obtenu le niveau de priorité le plus élevé après un accès bus, se verra attribuer le niveau de priorité le plus bas.
- **Niveau unique** SGL (*SinGle Level*)
  Les maîtres n'utilisent qu'un seul niveau de priorité, la ligne BR3*, pour demander l'accès au bus. Le niveau de priorité est lié à la position physique de chaque maître dans le châssis VME. Le maître le plus proche de l'arbitre sera donc prioritaire par rapport à un autre maître qui a émis une demande d'accès simultanée.

3 Les outils logiciels

La mise en place d'une architecture logicielle distribuée a été possible par l'utilisation des mécanismes de communication d'UNIX et de VxWorks. Nous présentons dans ce chapitre les outils choisis pour construire le système informatique de contrôle et commande qui assure la phase d'exploitation du Vivitron en insistant sur l'aspect programmation orientée objet. Ces outils se composent de l'exécutif temps réel VxWorks, du système de gestion de base de données orientée objet O_2 et du générateur d'interfaces graphiques orienté objet SL-GMS.

3.1 L'exécutif temps réel VxWorks

3.1.1 Définitions

Il n'existe pas de définition précise pour définir un système temps réel, mais on peut dire « *qu'un système fonctionne en temps réel à chaque fois qu'il sera question de contraintes de temps et que ces dernières seront respectées* » [11].

Un **noyau temps réel** ou **moniteur temps réel** offre les services minimums pour la mise en œuvre d'un système temps réel (des mécanismes de gestion de tâches, de synchronisation et de communication entre tâches). Cependant, la gestion de la mémoire et des Entrées/Sorties est prise en compte par le programmeur.

Un **exécutif temps réel** intègre un noyau et des fonctionnalités supplémentaires (gestion de la mémoire, des Entrées/Sorties, etc.) qui simplifient l'écriture des programmes et facilitent la maintenance du logiciel.

Un **système d'exploitation temps réel** offre les principaux services d'un exécutif, mais il fournit en plus des outils pour faciliter le développement des applications (compilateurs, par exemple). Il fournit un support logiciel complet favorisant la convivialité de l'interface homme-machine tandis qu'un exécutif est plus près du matériel dans un souci de performances et en dépit de la convivialité.

Un **processus** est un programme en exécution. Les différents processus d'un système s'exécutent en parallèle. Chacun a deux modes d'exécution possible : le **mode utilisateur** (le processus utilise ses propres ressources sans accéder au système) et le **mode superviseur** (le processus peut accéder au système et à des données privilégiées). VxWorks est un exécutif temps réel où toutes les tâches s'exécutent en mode superviseur, comme dans la plupart des exécutifs. Dans l'environnement VxWorks le terme de **processus** est remplacé par celui de **tâche** [22].

Un système de **commande de processus** doit effectuer les fonctions suivantes :

- agir sur les organes externes (lecture des capteurs et commande des actionneurs) ;
- prendre en compte le temps réel (traitement périodique et limitation du temps de réaction et d'exécution) ;
- réagir aux événements extérieurs (alarme, arrêt d'urgence, demande de service) ;
- gérer les informations (historique).

3.1.2 L'architecture générale

UNIX est un système d'exploitation qui n'est pas adapté aux applications temps réel. Il fournit une plate-forme de développement **multitâche**, **multi-utilisateurs** et **conviviale**. Il possède une interface de programmation réseau complète et offre des outils standard à l'utilisateur (éditeurs de texte, outils de développement, etc.). VxWorks est un exécutif temps réel **multitâche**, **mono-utilisateur**, créé par la société *Wind River System* [22]. Il est chargé de tester, mettre au point et exécuter les applications temps réel.

VxWorks utilise l'environnement UNIX pour développer ses applications temps réel (édition du programme source, compilation, etc.) et pour les fonctionnalités non temps réel (Interface Homme-Machine, interfaces réseaux extérieurs, etc.). La figure B.9 présente l'architecture globale de VxWorks.

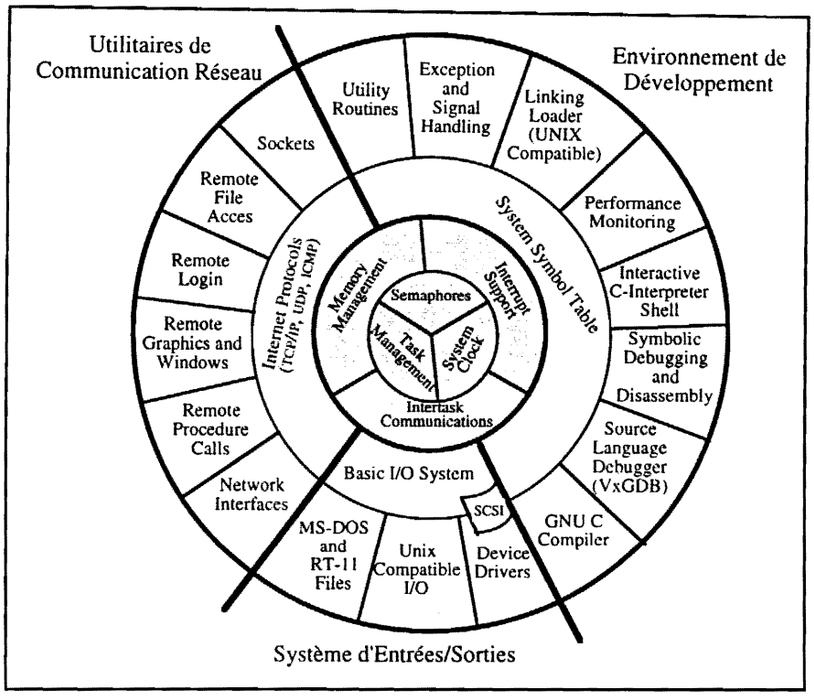


Figure B.9 - Architecture de VxWorks

VxWorks supporte la **réentrance** du code et évite ainsi sa duplication en mémoire. Il supporte aussi l'**architecture multiprocesseur** par l'intermédiaire du réseau. De plus, avec **WindX**, il offre un support pour les applications graphiques développées avec le système **X-Window**.

3.1.2.1 La gestion des tâches

Une tâche VxWorks est un programme en exécution qui utilise une mémoire divisée en trois segments :

- le **segment de texte** (*text*) pour le code objet ;
- le **segment de données** (*data*) pour les données statiques et dynamiques ;
- le **segment de pile** (*bss*) pour les informations temporaires (variables locales, paramètres des fonctions, etc.).

Chaque tâche est représentée par un descripteur de tâche, le **TCB** (*Task Control Block*), qui contient les informations essentielles à son exécution. Ces informations font partie du **contexte** de la tâche qui inclut :

Environnement machine

- compteur de programme **PC** (*Program Counter*) ;
- pointeur de pile **SP** (*Stack Pointer*) ;
- registres **CPU** (*Central Processing Unit*) ;
- registres à virgule flottante ;

Environnement système

- affectation des Entrées/Sorties standard (*stdin*, *stdout*, etc.) ;
- contexte du partage de temps ;
- structures de contrôle du noyau ;
- gestionnaires de signaux.

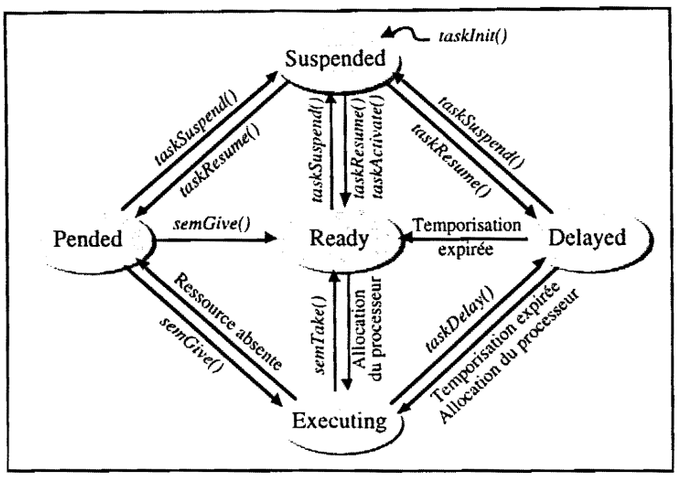


Figure B.10 - Diagramme d'état des tâches

Sous VxWorks, une tâche peut être :

- **En cours d'exécution** (*executing*)
  Etat d'une tâche qui dispose du processeur et de toutes les ressources dont elle a besoin.
- **Prête** (*ready*)
  Etat d'une tâche qui dispose de toutes les ressources dont elle a besoin sauf le processeur.
- **Bloquée** (*pended*)
  Etat d'une tâche qui est en attente d'une ressource (elle est bloquée sur un sémaphore, par exemple). Dans ce cas, elle n'est plus en possession du processeur.
- **Dormante** (*delayed*)
  Etat d'une tâche qui n'attend rien d'autre qu'un laps de temps défini.
- **Suspendue** (*suspended*)
  Cet état secondaire est utilisé pour la mise au point des tâches (aspect sûreté de fonctionnement et protection contre les erreurs). La tâche est suspendue volontairement par la routine *taskSuspend()*. Elle ne pourra être débloquée que par la routine *taskResume()*. Une fois débloquée, la tâche récupérera son état précédent sauf si cet état était l'état *executing*. Dans ce cas, elle prendra l'état *ready* et elle attendra que le processeur lui soit attribué.

La figure B.10 nous montre les différents états d'une tâche ainsi que les transitions entre états et les routines appelées lors de ces transitions. C'est la tâche *ready* de plus haute priorité (256 niveaux de priorité sont disponibles, le niveau de priorité le plus faible étant le numéro 255) qui sera en exécution (*executing*) à un instant donné.

Dans un système multitâche, les différentes tâches peuvent se trouver en concurrence pour obtenir le processeur. L'attribution de cette ressource commune est gérée par l'**ordonnanceur** (*scheduler*). L'exécutif VxWorks supporte l'**ordonnancement préemptif** : l'ordonnancement est basé sur la priorité préétablie des tâches ; ainsi, une tâche peut être interrompue (par le processus à contrôler : interruption matérielle) à tout moment pour être remplacée par une tâche de plus forte priorité, même dans le cas où la tâche à interrompre s'exécute à l'intérieur du noyau. Si deux tâches ont la même priorité, c'est la technique de la **priorité tournante** (*round-robin*) qui peut être utilisée : à chaque tâche est attribuée une **tranche de temps** ou **quantum** (*time slice*) pendant laquelle elle peut prendre le processeur ; lorsque sa tranche de temps est écoulée, le processeur est réquisitionné pour la tâche suivante dans la file d'attente.

3.1.2.2 Les communications entre tâches

Dans un environnement multitâche, les tâches ont besoin de coopérer pour s'échanger des données ou synchroniser leurs déroulements dans le temps, mais elles peuvent se trouver en concurrence pour obtenir des **ressources communes** (liste chaînée, zone mémoire, etc.). La partie de programme où l'on utilise des ressources communes est appelée **section critique**. L'accès à une section critique doit être exclusif et son déroulement indivisible : c'est le principe de l'**exclusion mutuelle**. Le système doit éviter qu'une tâche bloquée en dehors de la section critique empêche une autre tâche d'y accéder. Il doit s'assurer qu'il n'y ait pas d'**interblocage** (*deadlock*) entre deux tâches qui sont en attente de l'accès à la ressource critique.

Une des techniques pour assurer l'exclusion mutuelle est de masquer les interruptions par la routine *intLock()* avant d'accéder à la section critique, mais elle présente l'inconvénient d'allonger le temps de réponse aux interruptions. Une deuxième technique consiste à désactiver la préemption avec la routine *taskLock()* ce qui risque de bloquer une tâche de plus forte priorité. Ces deux moyens risquent d'augmenter considérablement le temps de réponse du système. Une solution consiste à utiliser un moyen logiciel de bas niveau qui signale l'utilisation d'une ressource commune : les **sémaphores**. VxWorks supporte trois sortes de sémaphores :

- **Les sémaphores binaires** (*semBcreate()*)
  Ils sont généralement utilisés pour résoudre les problèmes de synchronisation et d'exclusion mutuelle. Un sémaphore binaire est constituée de l'association d'une variable **verrou** et d'une **file d'attente**. Le verrou prend la valeur 0 quand la ressource est disponible et la valeur 1 quand la ressource est occupée. Une opération indivisible de verrouillage positionne le verrou à 1 (routine *semGive()*) et une opération indivisible de déverrouillage positionne le verrou à 0 (routine *semTake()*). Une tâche qui veut accéder à une ressource occupée est insérée dans la file d'attente et restera bloquée tant que la ressource ne lui sera pas attribuée. La file d'attente peut être gérée soit par **priorité** soit par **FIFO** (*First In First Out*). Un problème qui se pose avec ce type de sémaphore est que l'accès privé à une section critique n'est pas garanti. Un verrou qui a été positionné à 1 peut être positionné à 0 par n'importe quelle tâche.
- **Les sémaphores d'exclusion mutuelle** (*semMcreate()*)
  Ils sont adaptés particulièrement aux problèmes d'exclusion mutuelle. Le sémaphore d'exclusion mutuelle est une forme spéciale du sémaphore binaire. Seule la tâche propriétaire du sémaphore est autorisée à le libérer. L'unicité de l'accès à une section critique est ainsi garantie. Le

sémaphore d'exclusion mutuelle protège l'inversion de priorité, permet l'accès récursif à une ressource et protège la tâche d'une éventuelle destruction.

- **Les sémaphores à compte** (*semCcreate()*)
  Ils sont destinés à protéger les ressources qui peuvent être partagées par plusieurs tâches en même temps (réentrance). Ce type de sémaphore est une extension du sémaphore binaire. Le verrou est remplacé par un compteur qui fixe le nombre de détenteurs du sémaphore à un instant donné (maximum 256 détenteurs). Par exemple, si le compteur est initialisé à 2, deux tâches pourront accéder à la section critique sans se bloquer, mais la troisième sera bloquée et insérée dans la file d'attente. Dès qu'une des tâches fait appel à la routine *semGive()*, elle incrémente le compteur et relance une tâche bloquée.

Les trois types de sémaphores acceptent des attentes avec *timeout*. Les sémaphores sont utilisés dans la communication entre tâches locales.

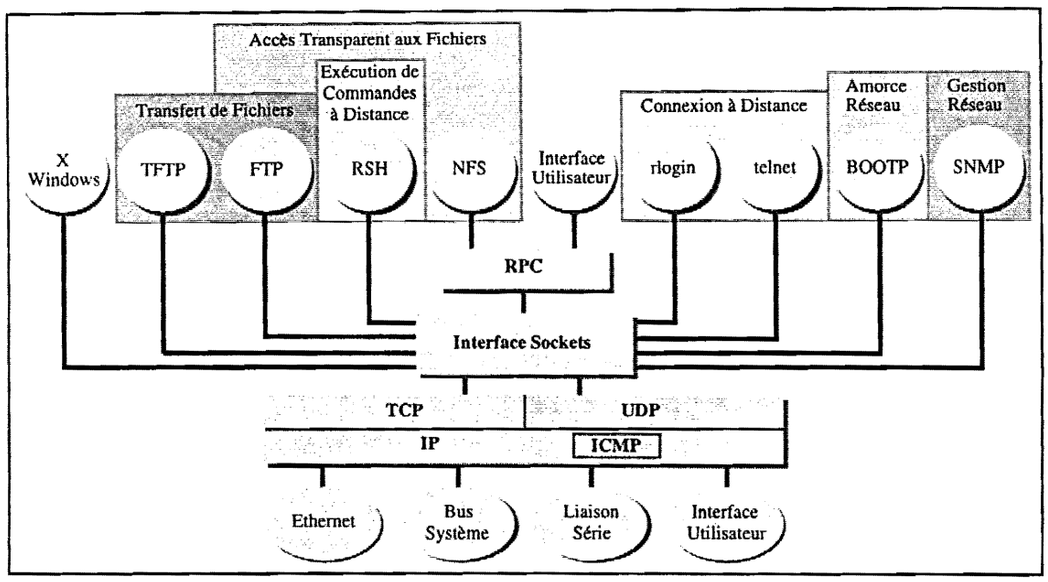


Figure B.11 - Les utilitaires VxWorks de communication réseau

La figure B.11 présente les différents utilitaires de communication réseaux supportés par VxWorks. Ils sont totalement compatibles avec UNIX 4.3 et utilisent les protocoles **Internet** pour les communications entre tâches distantes appartenant à deux systèmes VxWorks différents ou les communications entre une tâche d'un système VxWorks et un processus d'un Système UNIX (*sockets*, RPC, rsh). VxWorks supporte un riche ensemble de mécanismes de communication entre tâches locales ou distantes :

- La **mémoire partagée** (*shared memory*)
  Elle est utilisée pour le partage d'une zone de données commune sur le même processeur.
- Les **sémaphores** (*semaphores*)
  Ils sont utilisés dans les problèmes d'**exclusion mutuelle** (ressources non partageables), de sections critiques de codes et de synchronisation.
- Les **files de messages** ou **boîtes aux lettres** (*message queues*) et les **tubes** (*pipes*)
  C'est un mécanisme de communication entre tâches de plus haut niveau qui rend transparent au programmeur la gestion des sémaphores. Les tubes sont utilisés comme canaux de communication pour les files de messages (de type FIFO et de longueur variable).

- Les ***sockets*** et les **procédures à distance** (le protocole RPC)
  Ils permettent la communication entre tâches distantes à travers le réseau Ethernet.
- Les **signaux** (*signals*)
  Ils sont utilisés comme des **interruptions logicielles** lors du traitement d'**exceptions** (division par zéro, adresse incorrecte, erreur de bus, ...), ou pour la synchronisation entre tâches. Un signal est traité comme une interruption, c'est-à-dire qu'il déroute la tâche en exécution vers une routine de traitement du signal mais, contrairement à l'interruption, cette routine est lancée par une tâche. VxWorks dispose de 31 signaux et à chaque signal peut être connectée une routine de traitement différente.

3.1.2.3 La gestion des interruptions

Les interruptions matérielles (interruptions VME, erreur de parité en mémoire, panne d'alimentation...) sont des événements externes qui provoquent la suspension de la tâche en exécution et le passage à une routine de traitement de l'interruption (**ISR** *Interrupt Service Routine*). L'adresse de la routine ISR est associée à un **vecteur** identifiant le périphérique ou le circuit externe. Vxworks dispose d'outils qui permettent de manipuler les vecteurs d'interruption directement en langage C et d'écrire la routine ISR en langage C. Pour réaliser le traitement d'une interruption il suffit :

- d'écrire une fonction en langage C sans appel bloquant ;
- d'associer un vecteur à cette routine (*intConnect()*) ;
- d'autoriser l'interruption matérielle identifiée par le vecteur (*intUnLock()*);
- de réveiller une tâche lors du traitement de l'interruption en utilisant, par exemple, la synchronisation par sémaphores (figure B.12).

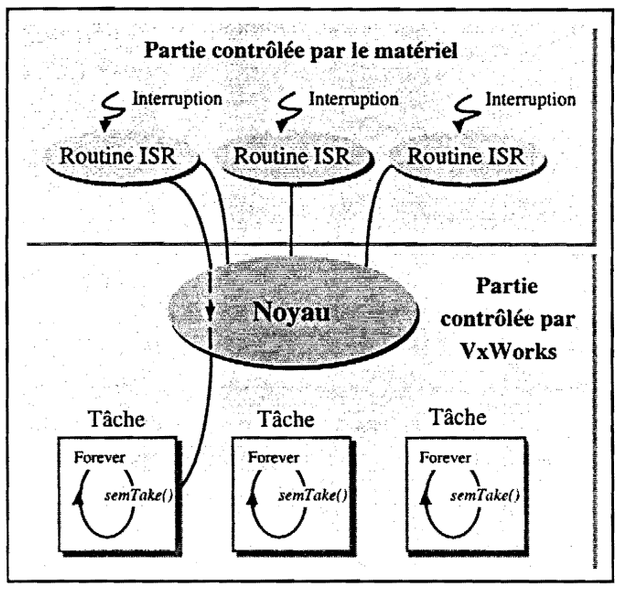


Figure B.12 - Les interruptions

En général, pour débloquer les tâches associées, les routines ISR utilisent des sémaphores, des tubes, des files de messages ou génèrent des signaux. Le **temps de latence** à uune interruption est inférieur à 10 µs. VxWorks dispose d'un mécanisme logiciel semblable aux interruptions : les **chiens de garde** (*Watchdog timers*). Ce mécanisme permet de connecter une routine écrite en langage C à l'expiration d'un délai. La routine associée au chien de garde doit obéir aux mêmes règles qu'une routine ISR.

3.1.3 Développement d'une application sous VxWorks

La mise en œuvre d'une application sous VxWorks demande une configuration minimale du système : une station de développement UNIX et un châssis VME contenant une carte processeur, le tout relié par réseau Ethernet. La carte processeur, ou encore cible VxWorks, contient le **noyau** et la **table de symboles standard** de VxWorks. Cette table contient, entre autre, les noms et adresses des fonctions et des variables globales utilisateur et système. Elle est très utile pour la mise au point des applications sous le *shell* VxWorks. La station de travail doit disposer d'un **compilateur croisé** C qui puisse générer des **modules objet** au format UNIX. L'édition de liens s'effectuera dynamiquement, lors du chargement des modules objets dans la cible VxWorks via le réseau Ethernet (commande **ld** *linker loader*).

L'accès à VxWorks s'effectue via le *shell* VxWorks. Celui-ci permet d'interpréter les expressions du langage C (appels aux fonctions, références aux variables) et permet au programmeur :

- d'appeler n'importe qu'elle routine du système VxWorks ou de sa propre application ;
- de connecter une routine à une interruption ;
- d'examiner et initialiser les variables d'une application ;
- de visualiser les fonctions et paramètres appelés par une tâche (trace de la pile de la tâche) ;
- de créer, suspendre, réveiller et changer la priorité des tâches ;
- de calculer le temps d'exécution d'une fonction ou d'un groupe de fonctions.

L'utilisateur communique avec le *shell* VxWorks depuis une console ou directement depuis le système UNIX. Il peut contrôler plusieurs systèmes temps réel VxWorks grâce au système multifenêtrage des stations de travail. Il dispose aussi d'outils de mise au point permettant de placer des points d'arrêt dans les programmes sources visualisés sur l'écran de la station.

3.2 La programmation par objets

3.2.1 Intérêts et motivations

La réalisation d'un logiciel se heurte, aujourd'hui, à une complexité de plus en plus grande de part sa taille et ses problèmes de modélisation et manipulation de l'information. Dans le domaine du **génie logiciel**, plusieurs problèmes se posent lors des phases de conception, réalisation et exploitation du logiciel :

- la **décomposition** du travail à réaliser ;
- la **coordination** de toutes les parties réalisées par les différents développeurs ;
- la **maintenance** du produit une fois fini ;
- la **réutilisation** du code pour d'autres applications.

Une solution à ces problèmes pourrait être l'introduction d'un plus **haut niveau d'abstraction** dans la modélisation de l'information et d'une **modularité** plus importante dans les méthodes de programmation. L'idéal serait un **Atelier de Génie Logiciel** (AGL) où le code serait généré automatiquement et ne serait plus écrit.

L'utilisation d'un langage machine comme l'assembleur dans la réalisation d'un logiciel rend ce dernier fortement dépendant du matériel. Il est surtout utilisé dans les applications temps réel telles que la robotique, les systèmes embarqués ou le contrôle de processus qui impliquent souvent la manipulation d'entités proches du matériel (adresses, registres, etc.). Cependant, la complexité grandissante des logiciels et la volonté de s'extraire des contingences matérielles de l'assembleur ont contribué au besoin de rechercher des langages plus évolués tout en offrant des fonctionnalités de bas niveau.

C'était le cas du langage Fortran qui apporta, outre la portabilité, la possibilité de manipuler des structures de contrôle de plus haut niveau. L'apparition de langages tels que C ou Pascal offraient une meilleure lisibilité que le Fortran et apportèrent la notion de **programmation structurée**. La programmation structurée est basée sur des **procédures** qui sont des bouts de programmes réutilisables, et des **structures de données** sur lesquelles les procédures agissent. Les données structurées réduisent considérablement le code et favorisent la modélisation d'une information complexe, mais atteignent leur limite dès lors que l'application évolue. De plus, une modification mineure dans la structure d'une donnée occasionne non pas une modification mineure du code source de l'application, mais une reprise globale de ce code. Les langages basés sur les principes **objets** tels que C++, successeur du langage C, résolvent ce dernier point.

Un **objet** est une structure complexe capable de modéliser par une donnée informatique une entité du monde réel. Il est doté de **données** accessibles uniquement par des **procédures** et cache au monde extérieur les détails de sa constitution interne. Toute modification de ses données n'aura des conséquences que limitées à lui-même et non à l'ensemble de l'application. La programmation par objets, avec un **plus haut niveau d'abstraction**, permet une représentation des données plus "naturelle" et le traitement de données plus complexes.

Cette nouvelle technologie **facilite la maintenance, la réutilisation du code et la portabilité du produit**, permet une méthode de travail plus rigoureuse et impose une analyse plus profonde du produit lors de sa phase de conception.

3.2.2 Le concept d'objet

L'idée majeure du concept objet est de conserver la même philosophie depuis l'analyse des besoins à partir du monde réel jusqu'à la réalisation de ces besoins sous forme d'objets. Un **objet** est doté d'**attributs,** agissant sur des **données** internes ou externes à l'objet et fournissant des services à travers des **méthodes**. Chaque objet est identifié par son **nom** (figure B.13).

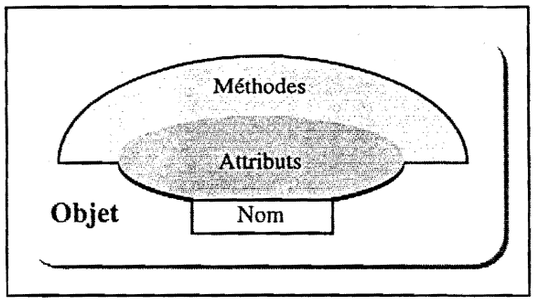


Figure B.13 - L'entité objet

Un objet regroupe un ensemble d'opérations et un état qui mémorise les résultats de ces opérations. Les objets communiquent entre eux par l'envoi de messages qui font appel à ces opérations. Les objets de même nature étant regroupés suivant la notion de **classe**, l'objet n'est en fait qu'un exemplaire de sa classe ; c'est ce qu'on appelle une **instance**. Deux instances d'une même classe se distingueront par les valeurs de leurs attributs.

Nous trouvons trois concepts importants associés aux objets qui favorisent l'écriture, la maintenance et la modularité des programmes :

- L'**héritage** est le mécanisme qui permet de définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute de nouvelles données et de nouvelles méthodes. La nouvelle classe ainsi créée **hérite** des propriétés de l'ancienne classe. Prenons deux classes, l'une appelée classe "mère" et l'autre classe "fille" ; si la classe "fille" hérite de la classe "mère", alors la classe "fille" est appelée **sousclasse** et la classe "mère" est appelée **superclasse**. L'héritage est dit **simple** si la sousclasse n'hérite directement que d'une classe ancêtre (superclasse). Il est dit **multiple** si la sousclasse hérite directement de plusieurs classes ancêtres.
- La notion d'**encapsulation** inclut, par ailleurs, la possibilité de cacher certains attributs à la vue du reste du programme. Un objet peut donc contenir des données invisibles aux autres objets et accessibles exclusivement par ses propres méthodes. L'encapsulation « *peut être considérée comme une généralisation de l'abstraction des données* » [23].
- Le **polymorphisme** correspond à l'idée qu'une même méthode peut aboutir à des comportements différents selon la classe d'objets à laquelle elle s'applique. Cela veut dire qu'il « *permet de définir plusieurs formes pour une méthode commune à une hiérarchie d'objets* » [23].

Pour illustrer le mécanisme d'héritage, l'exemple de la figure B.14 présente le graphe d'un ensemble de classes d'objets héritant les unes des autres. Un tel graphe est appelé **graphe d'héritage**. Puisqu'une voiture est un véhicule, la sousclasse ***Voiture*** hérite de tous les attributs et méthodes de la superclasse ***Véhicule***. Mais la classe ***Voiture*** peut également intégrer des attributs propres aux voitures, que ne possèdent pas les autres sousclasses de ***Véhicule***. Par exemple, la classe ***Voiture*** aura un attribut *coffre* qui ne concerne ni la classe ***Camion*** ni la classe ***Moto***. De même, la classe ***Camion*** aura un attribut *capacité_utile* (de chargement) qui ne concerne pas les autres classes. Tous les types de véhicules possèdent une vitesse maximale, une consommation et un prix ; les attributs *vitesse_maximale*, *consommation* et *prix* sont donc hérités par les trois sousclasses de **Véhicule**. La sousclasse **Voiture_de_Sport** hérite de ces mêmes attributs par l'intermédiaire de la classe **Voiture**.

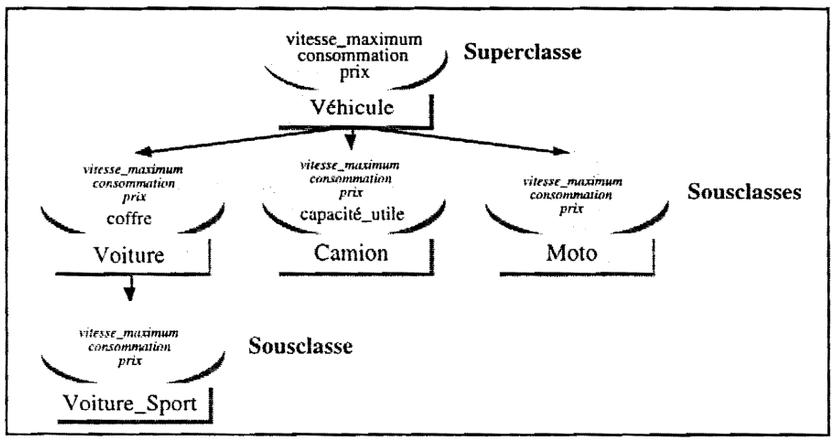


Figure B.14 - Graphe d'héritage simple

Ce mécanisme permet d'organiser les classes d'un système en un graphe d'héritage partant d'une classe générale pour arriver à des classes particulières plus complexes. Les classes sont donc à la fois génératrices de leurs instances et mères de leurs sousclasses. L'emploi judicieux de l'héritage est la clé du succès en programmation par objets. Il offre une approche descendante (*top-down*) et un raffinement graduel lors du développement d'un **schéma de classes d'objets.** Un **schéma** regroupe les définitions de tous les éléments qui composent une application orientée objet (classes, méthodes, noms d'objet, etc.). L'héritage évite la répétition du code en facilitant la réutilisation, permet de minimiser les modifications de code et permet une modélisation plus compacte et mieux structurée.

3.2.3 Les bases de données objet

Les **Systèmes de Gestion de Base de Données** (SGBD) sont des outils informatiques qui offrent à un groupe d'usagers la possibilité de partager, de traiter et d'interroger une grande quantité de données en garantissant leur fiabilité (cohérence, intégrité, confidentialité et sécurité) [24]. Ces données, stockées généralement sur des supports magnétiques, « *sont dites persistantes car elles restent en général sur leur support même si le programme qui a servi à les créer ne "tourne" plus* » [24].

Les systèmes relationnels, apparus au début des années 1980, sont les plus utilisés aujourd'hui, mais ils sont vite limités dans les nouvelles applications qui manipulent, de plus en plus, des objets complexes (textes, images, sons, graphiques). Pour remédier à cette défaillance, les fonctionnalités des SGBD ont été fusionnées à celles de la programmation orientée objet. De cette fusion sont nés les **Systèmes de Gestion de Base de Données Orientés Objet** (SGBDOO). Un SGBDOO doit satisfaire à deux critères : il doit être un SGBD et il doit intégrer les éléments de la technologie objet suivants :

« • *description d'objets complexes supportant le typage et l'encapsulation ;*
- *identification des objets permettant leur partage, leur copie et leur renommage ;*
- *persistance sélective des objets ;*
- *manipulation de collections d'objets* » [25].

Un SGBDOO apporte les avantages suivants :

- Pour la manipulation d'objets complexes, le modèle relationnel devient trop lourd à gérer et est très vite limité. En revanche, le modèle objet est beaucoup mieux adapté à ce type de problème, et permet de traiter des représentations de données plus complexes.
- Avec un plus haut niveau d'abstraction, la représentation des données devient plus "naturelle".
- Il facilite la maintenance, la réutilisation du code et la portabilité du système.
- Sa prise en main est facile.

Un SGBDOO présente les inconvénients suivants :

- Pour des objets simples, il est difficile d'obtenir des performances identiques au modèle relationnel compte tenu de la richesse d'information nécessaire pour la technologie objet.
- Les produits qui se trouvent sur le marché sont encore jeunes ce qui fait qu'aucun d'entre eux n'a pu s'imposer comme un standard.
- La mise à jour du schéma est difficile.

3.3 La base de données O2

Le système O_2 est issu d'un projet de Recherche et Développement (R&D) au sein du groupement ALTAÏR, où l'on retrouve l'INRIA, l'IN2 et le LRI. Ce groupement a donné naissance à la société O_2Technology qui commercialise le produit O_2 depuis 1990. Pour répondre aux exigences des utilisateurs, le produit O_2 n'a pas cesser d'évoluer et de s'enrichir d'outils nouveaux. C'est pourquoi, nous sommes restés en étroite collaboration avec le service technique de la société O_2Technology tant pour répondre aux problèmes rencontrés que pour corriger les défauts apparus en cours d'évaluation du produit.

3.3.1 L'architecture générale

Le système O_2 est un système de base de données orienté objet, distribué, muni d'un environnement complet de programmation [26]. Cet environnement intègre un langage de quatrième génération orienté objet, un langage de requête orienté objet, et un générateur d'interfaces graphiques qui nous donne une Interface Homme-Machine (IHM) à moindre coût.

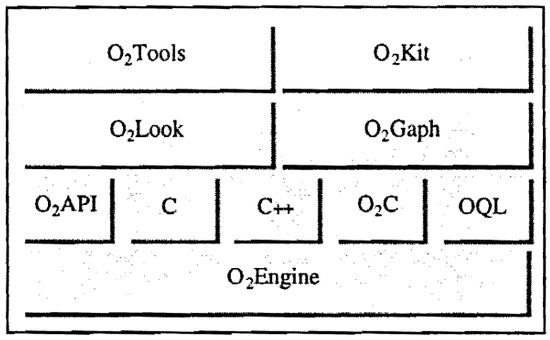


Figure B.15 - Architecture fonctionnelle de O_2

Le SGBD O_2 fournit un ensemble de modules [26] reposant sur un noyau de gestion d'objets : **O_2Engine**. Ce noyau est basé sur une architecture matérielle de type **client/serveur** et utilise le protocole **RPC** comme moyen de communication. Un serveur (**O_2server**) permet l'accès simultané de plusieurs clients aux données manipulées dans la base. Il gère lui-même son espace disque, les zones de *swapp*, les caches, les accès concurrents par plusieurs utilisateurs, la persistance des objets créés (stockage des objets sur disque), la sécurité des transactions effectuées, les reprises sur panne... O_2Engine fournit les notions d'un système à objets (classes, méthodes, héritage simple et multiple, encapsulation...) et supporte plusieurs interfaces de programmation (figure B.15) :

- **O_2API** *(Application Programming Interface)*, une interface qui offre un accès direct à O_2Engine (manipulation de classes, d'objets complexes...).
- Deux interfaces de programmation pour les langages **C** et **C++**.
- **O_2C**, un langage de 4ème génération qui est, en fait, une extension du langage C contenant les fonctionnalités correspondant à la manipulation des objets O_2. Il permet la création d'objets O_2, l'emploi de messages et la manipulation de données complexes. O_2C dispose d'un compilateur incrémental et d'un éditeur de liens dynamique.

- **OQL**, un langage de requêtes orienté objet pour rechercher des objets dans une base. Il combine les possibilités d'interrogation des langages relationnels (SQL *Structured Query Language*) avec la technologie objet. Il peut être inclus dans le langage O_2C, facilitant ainsi l'expression de certaines recherches.

Le SGBD O_2 offre aussi les outils suivants :

- **O_2Look**, un générateur automatique d'interfaces utilisateur qui permet l'écriture rapide de l'interface d'une application et une manipulation aisée des présentations graphiques d'objets.
- **O_2Graph**, un module pour créer, modifier et éditer toute sorte de graphes.
- **O_2Tools**, un environnement graphique complet de programmation, construit autour du système X-Window, qui permet la navigation et l'édition dans le schéma et la base de données, la mise au point des méthodes, les tests...
- **O_2Kit**, un ensemble de classes O_2 importables dans d'autres schémas et pouvant être personnalisées (des dates, du texte, des boites de dialogue, etc.).
- **O_2Xt**, un module pour créer des interfaces graphiques avec les librairies X.

3.3.2 Le système O2

3.3.2.1 Le modèle de données

O_2 permet de définir des **types** et des **valeurs**. Une **valeur** est une donnée qui possède un type. Le **type** définit la structure de la donnée (exemple : *integer* définit le type entier) ainsi que les opérations qu'on peut lui appliquer (exemple : +, -, *, /). Un type est construit à partir de 6 types non sécables (atomiques) :

- **integer**, pour les entiers ;
- **character**, pour les caractères ;
- **boolean**, pour les valeurs logiques ;
- **real**, pour les valeurs réelles ;
- **string**, pour les chaînes de caractères ;
- **bits**, pour les valeurs binaires.

O_2 nous offre également 3 types de données structurées :

- **tuple**, pour décrire les n-uplets ;
- **set**, pour les ensembles désordonnés de valeurs ou d'objets de même type ;
- **list**, pour les listes ordonnées de valeurs ou d'objets de même type.

O_2 permet de définir des **classes** et des **objets**. Une **classe** est constituée de son type et de ses méthodes ; un **objet** est une donnée qui est associée à une classe. La classe définit la structure de l'objet ainsi que les **méthodes** qui peuvent lui être appliquées. L'**encapsulation** concerne aussi bien les attributs d'une classe que les méthodes. O_2 dispose de trois niveaux de visibilité :

- **privé** (*private*) : seules les méthodes de la classe A peuvent accéder aux attributs ou aux autres méthodes de la classe A ;
- **lecture** (*read*) : ce niveau de visibilité ne s'applique qu'aux attributs (l'attribut de la classe A n'est accessible qu'en lecture en dehors de la classe A) ;
- **écriture** (*public*) : toutes les méthodes (internes ou externes à la classe A) peuvent lire ou modifier les valeurs des attributs de la classe A et accéder aux méthodes de la classe A.

La figure B.16 nous donne un exemple de classe et d'objet O_2. Dans cet exemple, nous avons encapsulé tous les attributs (attributs privés) sauf l'attribut *nbr_habitants* qui lui est public. Il n'est donc pas possible d'accéder directement au nom, à la superficie ou à la langue d'un pays ; pour toute modification il faudra obligatoirement passer par des méthodes (*changer_nom*, *changer_superficie*, *changer_langue*). Le nombre d'habitants peut, au contraire, être modifié directement.

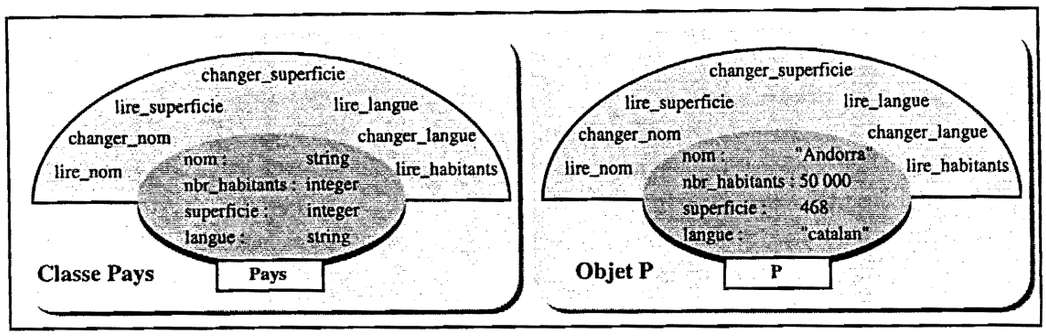


Figure B.16 - Exemple d'une classe et d'un objet O_2

La **persistance** est gérée en O_2 par la déclaration de **racines de persistance**. Tout objet ou valeur attaché à une racine de persistance obtient automatiquement tous les bénéfices offerts par le système de bases de données. D'autre part, O_2 supporte l'héritage simple et l'héritage multiple. Une classe prédéfinie appelée ***Object*** constitue la racine du **graphe d'héritage** de O_2.

3.3.2.2 La structure du système O2

Pour mettre en place une base de données O_2, la première étape consiste à créer un schéma. Le **schéma O_2** définit la structure logique de la base de données. Il est constitué d'un ensemble de définitions de classes, d'objets, de racines de persistance et d'applications. A ce schéma nous pouvons associer plusieurs applications (ensemble de programmes) ainsi que une ou plusieurs bases indépendantes. Une **application O_2** est un ensemble de programmes dont les objets créés sont stockés dans la base. Une **base O_2** est uniquement constituée de valeurs et d'instances dont les classes sont définies dans le schéma associé. L'ensemble des schémas, des bases et des applications est organisé autour du **système O_2** (figure B.17).

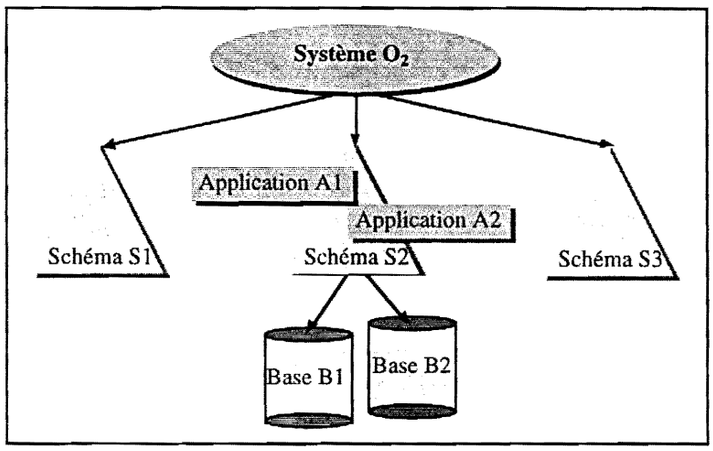


Figure B.17 - Le système O_2

3.4 L'interface graphique GMS

Le produit SL-GMS (*Sherrill Lubinski - Graphical Modelling System*) est développé aux Etats Unis par la société SL Corporation et commercialisé en France depuis 1992 par la société TENET Systems, filiale de SIGMEX. Il est utilisé par les développeurs pour doter leurs applications d'écrans graphiques animés. Ces derniers sont de plus en plus souvent utilisés dans les applications qui gèrent le temps réel. SL-GMS simplifie la construction de ce type d'écrans afin d'assurer le contrôle et l'affichage du comportement dynamique de telles applications.

3.4.1 L'architecture générale

SL-GMS est un générateur d'interfaces graphiques animées et interactives. Son architecture est basée sur une méthodologie orientée objet et s'intègre pleinement dans le système de fenêtrage X-Window. Il est muni d'un environnement de développement complet qui offre toutes les fonctionnalités nécessaires pour la construction d'écrans graphiques complexes (manipulation de la taille et de la position, modification du comportement graphique, etc.) et pour l'interaction avec ces écrans pendant la phase d'exploitation. Il se compose essentiellement de [27] :

- **DRAW**, un éditeur graphique interactif qui permet de créer des objets graphiques, de leur donner des propriétés dynamiques et de les tester ;
- **GML** (*Graphical Modelling Language*), un interpréteur de commandes interactif qui permet de créer des objets graphiques, de leur associer un comportement dynamique et de les visualiser ; il est utilisé comme un complément de DRAW ;
- **GMF** (*Graphical Modelling Function*), une bibliothèque de fonctions graphiques pour l'animation des écrans ;
- **SMS** (*Screen Management System*), un gestionnaire d'écrans qui permet le prototypage directement pendant la phase de création des écrans graphiques ;
- **GMD** (*Graphical Modeling Dynamics*), une interface entre les variables de l'application et les objets graphiques ;
- **GMSRUN**, un module d'exploitation qui lit les fichiers de description des écrans ("*modèle.ml*") et gère leur affichage (rafraîchissement périodique). En mode test, GMSRUN peut être utilisé comme un outil indépendant pour la visualisation dynamique ou statique des écrans. En mode exploitation, il peut être noyé dans l'application utilisateur.

GMS sépare entièrement la partie graphique de la partie applicative, et le lien entre les deux se fait par des **variables dynamiques**. La partie graphique est stockée dans des fichiers de ressources ("*modèle.ml*") lus par l'application lors de son démarrage. C'est à ce moment que les variables de l'application sont connectées aux éléments graphiques (figure B.18).

Un **écran GMS** est modélisé par un **modèle GMS**. Un **modèle** est défini par un ensemble d'objets graphiques (graphes, icônes, lignes, rectangles, etc.) appelés **sous-modèles**. Le sous-modèle est l'élément de base du modèle. A un sous-modèle on peut lui appliquer des **propriétés dynamiques** (attributs des objets graphiques) qui contiennent des actions (changement d'échelle ou de couleur, activation de fenêtres, etc.) et des **variables dynamiques**. Un changement de valeur d'une variable dynamique provoquera le déclenchement des actions associées. Ces sous-modèles sont appelés **sous-modèles dynamiques**.

Les sous-modèles dynamiques qui interagissent avec la souris ou le clavier sont appelés **GISMOS** (*Graphical Interactive Screen Management Objects*). Un **GISMO** peut être vu comme un *widget* de la *Xtoolkit* (§1.3.3) auquel on lui a rajouté plus de fonctionnalités. GMS dispose d'une bibliothèque standard de GISMOS que le développeur peut agrandir en créant ses propres GISMOS (le curseur de nos applications GMS (chapitre C, §6.1.1)). Les GISMOS sont des boutons, des curseurs, des interrupteurs... et peuvent être utilisés en tant qu'actionneurs pour contrôler, par exemple, les processus temps réel.

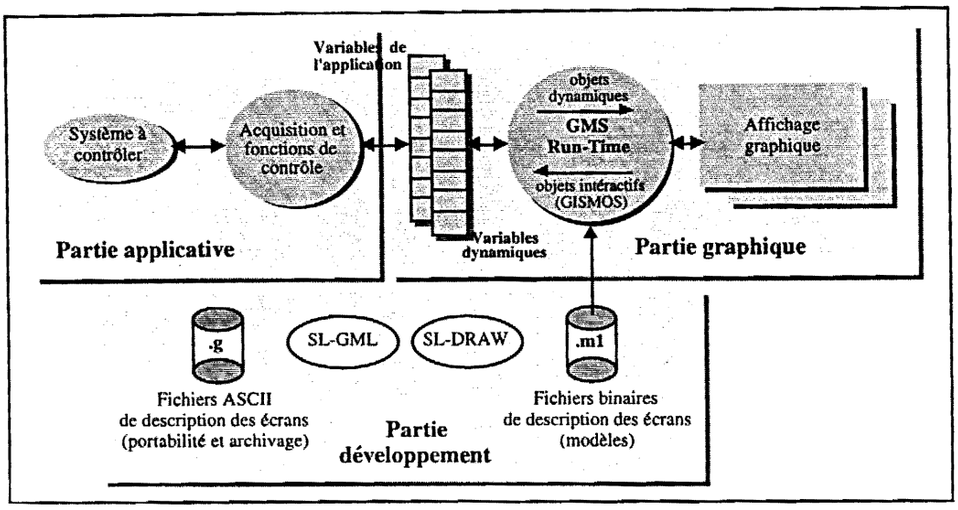


Figure B.18 - Fonctionnement de GMS

Dans un environnement multifenêtrage, une application GMS peut être composée de plusieurs écrans graphiques (modèles) qui s'affichent simultanément sur un écran physique. Sous le système X-Window, les écrans GMS sont des fenêtres "filles" de la *root window* (§1.3.3.2), appelée *Workstation Window* sous GMS. La position de ces fenêtres sur la *Workstation Window*, la partie visible de chaque fenêtre et la liste de toutes les fenêtres appartenant à l'application GMS sont contenus dans une vue (***view***). Les renseignements sur la *Workstation Window*, la view et la liste des modèles qui composent une application GMS sont contenus dans le comportement graphique de l'application : le ***state*** (**état**).

3.4.2 La structure d'une application GMS

GMS fournit au développeur une librairie de fonctions (**libgms**) pour la programmation d'une application GMS en langage C ou C++. La structure générale du code d'une application GMS comprend un module principal contenant le sous-programme *gms_main()* et d'autres modules pour les classes de *states*, les instances de *states*, les variables dynamiques et les fonctions utilisateur. Nous ne verrons ici que le module principal qui est constitué de la façon suivante :

- **Initialisation de GMS**
  - *gmsSetup()*, initialise GMS et appelle les fonctions d'initialisation utilisateur ;
  - *gmsMainInit()*, crée la *Workstation Window* et initialise les GISMOS ;
  - *gmsInitStates()*, initialise le mécanisme des *states*.

- **Initialisation de l'application**
  Initialisation des variables dynamiques ; création et lancement de states ; définition d'un rythme de rafraîchissement des écrans GMS...
- **Activation de la boucle principale**
  gmsMainLoop(), boucle de gestion des événements X et de la mise à jour des variables dynamiques.

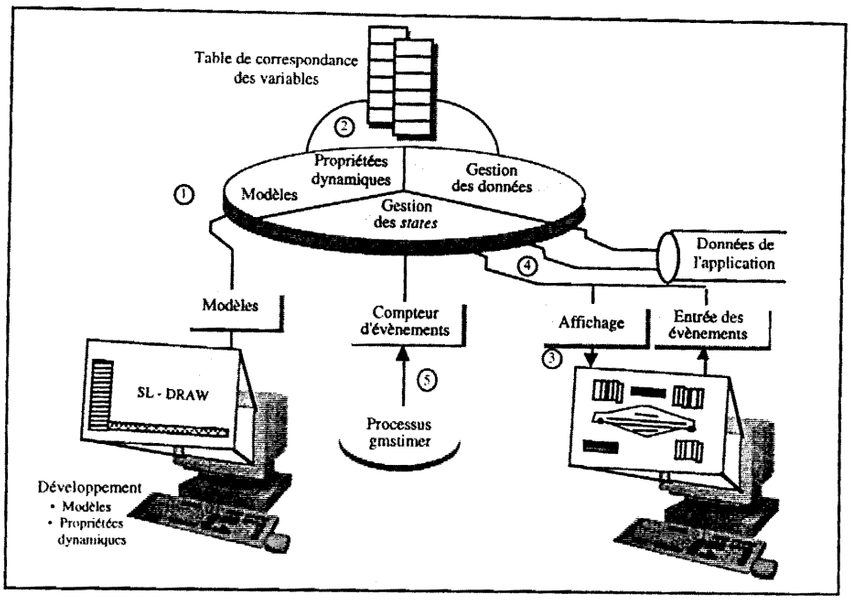


Figure B.19 - Application SL-GMS

La figure B.19 nous montre le déroulement d'une application GMS. Celle-ci intervient sur les modèles (écrans) (1) créés avec l'éditeur DRAW, qui modifie leur apparence en fonction de l'évolution des variables de l'application au travers d'une table de correspondance des variables (2). L'affichage graphique évolue en fonction de l'arrivée d'événements utilisateurs (click sourie) et des données de l'application (4). Un compteur d'événements fournit par le biais d'événements X "l'horloge" cadençant la mise à jour de l'affichage (5).

3.4.3 La particularité de GMS

La grande particularité de GMS est son **approche monolithique qui rend difficile la modularité d'une application utilisateur**. GMS utilise le système multifenêtrage X-Window, et donc une programmation par événements X (§1.3.3). Le programme est construit autour d'une boucle principale (*gmsMainLoop*()) dont la fonction essentielle est de se mettre en attente d'événements envoyés par le serveur X (figure B.8, §1.3.3.4). Lorsqu'un événement survient, le programme analyse la nature de celui-ci et réagit en conséquence. GMS possède un processus indépendant qui envoie périodiquement un événement X de type **gmstimer** à l'application GMS (figure B.19). Dès réception de ce type d'événements, GMS exécute une séquence où il traite toutes les fonctionnalités liées à son application (modification des variables dynamiques, rafraîchissement des modèles graphiques...). C'est dans cette boucle que l'utilisateur pourra insérer les traitements liés à son application.

4 Contrôle et commande

4.1 Introduction

L'activité de l'équipe de Contrôle et Commande s'est exercée dans deux directions : la mise en œuvre d'un système permettant la conduite des essais en tension du générateur et son évolution vers un système de contrôle définitif pour la phase d'exploitation de l'accélérateur. L'architecture matérielle et logicielle du système de contrôle et commande va être validée pendant la phase de test [1]. Le système matériel, décrit sur la figure B.20, comporte trois niveaux :

- les **capteurs et actionneurs** près des équipements ;
- les **ordinateurs frontaux** (dont certains sont embarqués, ceux des extrémités et du centre de l'accélérateur et ceux de l'injecteur) réalisés au standard VME, rassemblant et filtrant les informations et assurant certaines prises de décision ;
- les **stations de travail** présentant les informations sous forme graphique, signalant les alarmes ou permettant de commander des interventions sur les équipements.

Ce système est distribuée sur un **réseau Ethernet**. Les liaisons entre l'intérieur et l'extérieur de la machine s'opèrent par **fibre optique** depuis l'injecteur et les extrémités, et par **faisceau laser** depuis le centre, toute liaison galvanique étant exclue.

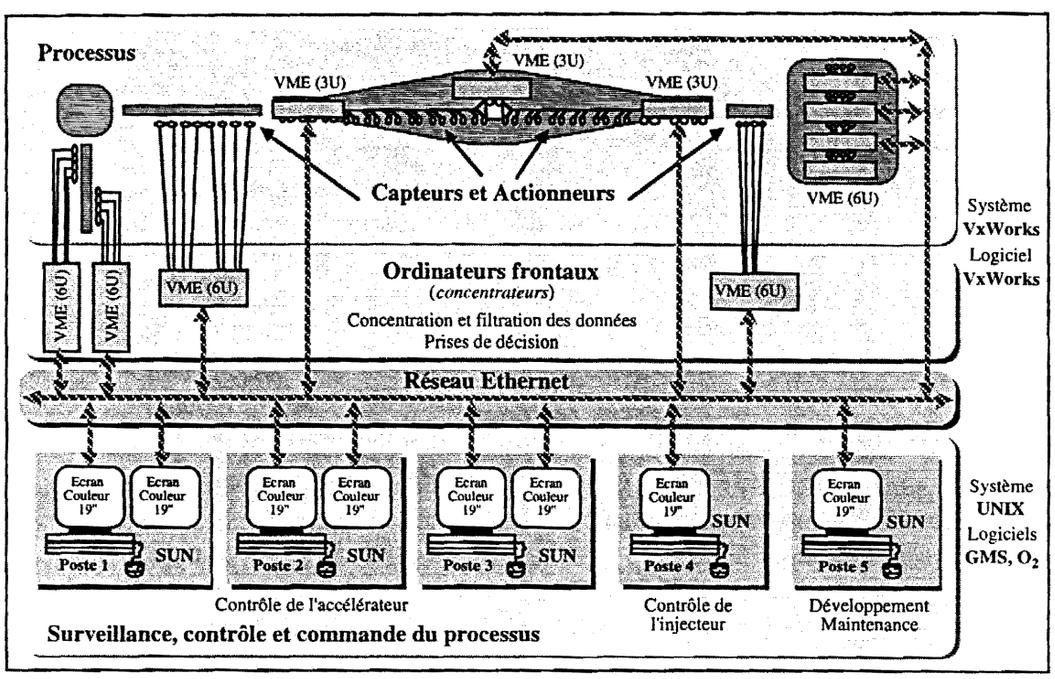


Figure B.20 - L'architecture matérielle et logicielle du système de contrôle et commande

Le CRN dispose d'un parc assez important de serveurs et de stations de travail fonctionnant sous le système UNIX. C'est pourquoi il a été décidé d'utiliser les stations de travail SUN, où s'intègre

pleinement l'environnement UNIX, pour le développement et l'exploitation de l'architecture logicielle du système. Un accélérateur comme le Vivitron sera surveillé, contrôlé et commandé à partir d'un pupitre comportant trois postes de travail composés chacun d'une station de travail SUN équipée de deux écrans. Un troisième et quatrième poste, composés d'un seul écran, seront dédiés respectivement au développement et au contrôle séparé de l'injecteur (figure B.20).

Les choix informatiques ont été faits lors de l'inauguration de la phase d'exploitation. Les outils logiciels adoptés sont : l'exécutif temps réel VxWorks déjà utilisé pour l'acquisition de données des multidétecteurs (EUROGAM, DIAMANT, DEMON, ICARE, etc.), la base de données orientée objet O_2 et le générateur d'interfaces graphiques animées orienté objet SL-GMS.

4.2 L'architecture matérielle

L'architecture matérielle est conçut sur une structure en couches (figure B.21) :

- La **couche haute**, regroupant les stations de travail qui intègrent la visualisation des informations, les actions sur les équipements et la mémoire du système (livre de bord, historique, etc.).
- La **couche intermédiaire**, composée d'ordinateurs frontaux dotés "d'intelligence" pour concentrer et filtrer les informations et pour assurer les prises de décision.
- La **couche basse**, regroupant les interfaces physiques avec le processus (capteurs et actionneurs près des équipements) fonctionnant souvent dans des conditions difficiles (perturbations électromagnétiques, risques de claquages, etc.). Cette couche assure l'acquisition et la commande des données ainsi que leur aiguillage vers la couche intermédiaire.

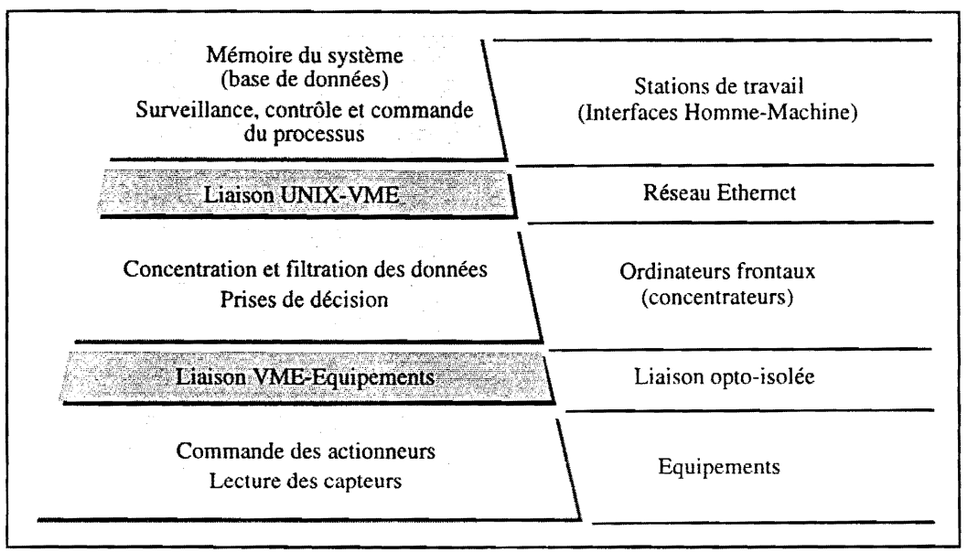


Figure B.21 - Architecture matérielle

Les transmissions des données entre les différentes couches doivent être assurées par un support isolé, rapide et offrant un débit d'informations acceptable pour un système de contrôle et commande

fonctionnant en temps réel. La nature et les caractéristiques des liaisons doivent être définies au cas par cas, en essayant d'utiliser des standards et de conserver une grande homogénéité. La plupart des équipements se trouvant à des potentiels électriques différents (à la masse autour de la machine et à des potentiels variables dans la machine et dans l'injecteur) (chapitre A, §1.4.1), le transfert de l'information par voie optique s'impose (fibres optiques, lasers ou faisceaux de lumière infrarouge). Toutefois, la liaison galvanique est adoptée lorsque la liaison optique n'est pas utilisable par la nature du capteur ou de l'actionneur. C'est le cas des liaisons dans les zones à champ faible. Elles établissent la communication entre capteurs, actionneurs et interfaces selon des techniques de blindage et de protection rigoureuses (double blindage, canalisation coaxiale, etc.). Les liaisons optiques se décomposent en trois catégories :

- **Les liaisons standard**
  Ces liaisons sont distribuées sur le seul réseau standard, le réseau Ethernet. Les liaisons entre l'intérieur et l'extérieur de la machine s'opèrent par fibre silice depuis l'injecteur et les extrémités et par faisceau laser depuis le terminal.
- **Les liaisons VME-équipements**
  Les liaisons entre les automates, ou les cartes d'interface, et les équipements (capteurs de courant, interrupteurs optiques, etc.) s'opèrent par fibre optique sans gaine ou par des faisceaux de lumière infrarouge.
- **Les liaisons particulières**
  Pour la commande et la mesure de la position des vannes, nous utilisons des fibres optiques.

Les réseaux de terrain (BITBUS, FIELBUS, CAN, PROFIBUS, etc.) offrent, il est vrai, l'avantage d'être bon marché et fiables mais, malheureusement, ils n'offrent aucun véritable standard. En effet, nous ne trouvons que des solutions constructeurs ou de consortium constructeurs. Ils sont relativement lents, nécessitent des gestionnaires d'interfaces et nous interdisent de disposer "d'intelligence" au plus près des équipements. Nous voulons aussi avoir la possibilité d'effectuer un (ou des) contrôle(s) local (locaux) performant(s). Nous avons choisi de ne pas utiliser des bus de terrain mais de mettre de "l'intelligence" au plus près des équipements, ce qui impose pratiquement l'amenée d'Ethernet jusqu'à chaque ordinateur frontal.

4.2.1 Les concentrateurs

Pour la phase d'exploitation de l'accélérateur, nous avons décidé de placer des ordinateurs frontaux sans écrans à proximité des points de contrôle ou des périphériques. Ces ordinateurs, qu'on appellera **concentrateurs**, sont dotés "d'intelligence" pour assurer certaines prises de décision, et pour concentrer et filtrer les données. Ils sont connectés entre eux par réseau Ethernet et les résultats sont rassemblés vers des stations de travail qui présentent les informations sous forme graphique, signalant les alarmes ou permettant de commander des interventions sur les équipements, ce qui débouche à un **système distribué**.

La moitié des paramètres étant situés à un potentiel différent de la masse, il est nécessaire d'inclure plusieurs *concentrateurs* à l'intérieur de l'enceinte accélératrice et dans l'injecteur. Pour une raison de standardisation, nous avons utilisé, là où c'était possible, des châssis VME au format **double europe** contenant une carte processeur MVME 167 Motorola à base d'un processeur MC68040 cadencé à 25 MHz. En revanche, vus les problèmes d'encombrement à l'intérieur de la cuve, notre choix s'est porté sur l'utilisation de châssis VME au format **simple europe** avec une carte processeur VM40 PEP

à base d'un processeur MC68040 25 MHz. Les limitations dues au **simple europe** (taille et performances) sont acceptables, d'autant plus que des noyaux Temps Réel du type VRTX et VxWorks commencent à apparaître sur de telles cartes, pour réaliser des applications de même performance qu'en double europe.

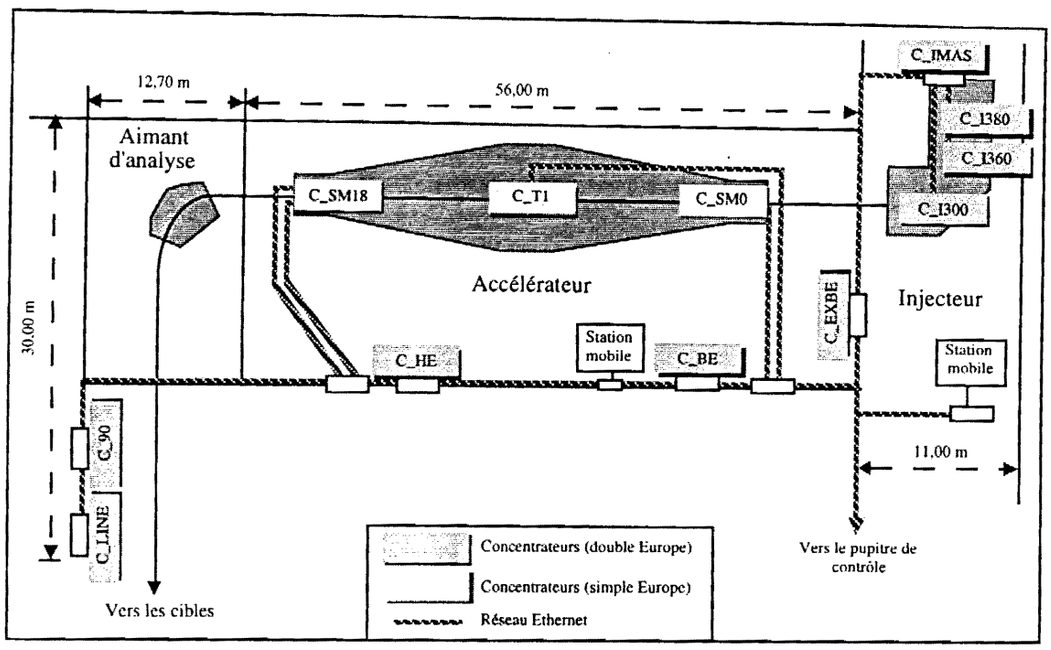


Figure B.22 - Emplacement des concentrateurs

La figure B.22 nous montre l'emplacement des *concentrateurs* auprès du Vivitron. Les informations concernant l'extérieur de la cuve de l'accélérateur sont regroupées dans le *concentrateur* situé à l'**EXtrémité Basse Energie** (C_EXBE). Celles concernant l'intérieur de la cuve sont regroupées dans les concentrateurs du Terminal et des **Sections Mortes** 0 et 18 (C_T1, C_SM0 et C_SM18). Quatre *concentrateurs* contrôlent les paramètres de l'**injecteur** (§1.3.2.1), chacun étant placé à un potentiel électrique différent : la masse pour C_IMAS, 300 kV pour C_I300, 360 kV pour C_I360, et 380 kV pour C_I380. Le contrôle du faisceau aux deux extrémités de l'accélérateur, **Basse Energie** et **Haute Energie**, est assuré par deux *concentrateurs* extérieurs (C_BE et C_HE). L'aimant d'analyse 90°, à la sortie de l'accélérateur, et la **ligne du faisceau** jusqu'aux cibles du physicien (§1.3.3.2) sont contrôlés par les *concentrateurs* extérieurs C_90 et C_LINE respectivement.

Ces *concentrateurs* sont équipés de diverses cartes d'interface pour réaliser les acquisitions ou le contrôle des données, ainsi que de quelques automates programmables utilisés essentiellement pour gérer les dipôles magnétiques ou les pompes à vide.

4.2.2 Les cartes d'interfaces

L'équipe de contrôle et commande a défini des normes dues aux équipements et aux contraintes de la machine :

- les interfaces pour les mesures et les commandes sont isolées **galvaniquement** ;

- les commandes et les mesures analogiques se font sur une plage variant de 0-10 volts puisque la plupart des équipements répondent à ces caractéristiques de fonctionnement. Les équipements hors normes sont systématiquement modifiés ou une interface d'adaptation spécifique est construite ;
- utilisation d'une méthode maison d'adressage VME ;
- utilisation d'un minimum de cartes d'interfaces différentes, ce qui nous a conduit à retenir les 6 modèles présentés sur le tableau T.6

Interface	Entrées/Sorties	Type
VDAC	Sorties	Analogique
VADC	Entrées	Analogique
VMOD	Entrées/Sorties	Tout ou Rien
XYCOM	Entrées	Module de comptage
MOTEUR	Entrées/Sorties	Pulsation
GPIB	Entrées/Sorties	Pulsation

Tableau T.6 - Les différents modèles de cartes d'interfaces

4.2.2.1 Les convertisseurs digitaux-analogiques VDAC12

Ces convertisseurs digitaux-analogiques [28] permettent de convertir un signal numérique sur 12 bits en un signal analogique variant de 0 volt à 10 volts. Le temps d'établissement, après une demande de conversion, est de 7 µs pour la pleine échelle. Chaque DAC comporte 8 voies en sortie. Toutes les voies sont isolées galvaniquement (500 V DC). Tension et intensité d'utilisation : 5V ; 0,7A.

4.2.2.2 Le convertisseur analogique-digital VADC23

Ces convertisseurs analogiques-digitaux [29] permettent de coder un signal analogique 0-10 volts en entrée sur 12 bits (15 µs de temps de conversion pour un gain de 1). Le gain en entrée peut être de 1, 10, 100 ou 1000. Chaque convertisseur est équipé de deux cartes filles comportant chacune 16 voies d'entrées unifilaires ou 8 voies différentielles.

4.2.2.3 La carte Tout ou Rien VMOD

Ces cartes Tout Ou Rien [30] assurent toutes les mesures et commandes logiques et peuvent être équipées de deux cartes filles du type :

- PB-DIN Digital Input 16 Entrées.
- PB-DIO Digital I/O 8 Sorties (5V 10mA) et 8 Entrées.

4.2.2.4 Le convertisseur fréquence-courant XYCOM

Ces convertisseurs à 8 entrées (multifonction, programmable par logiciel) [31] sont associés, dans notre cas, à des capteurs de courant émettant des impulsions dont la fréquence est proportionnelle au courant. Cette interface associée au programme d'acquisition a été programmée en mode fréquencemètre.

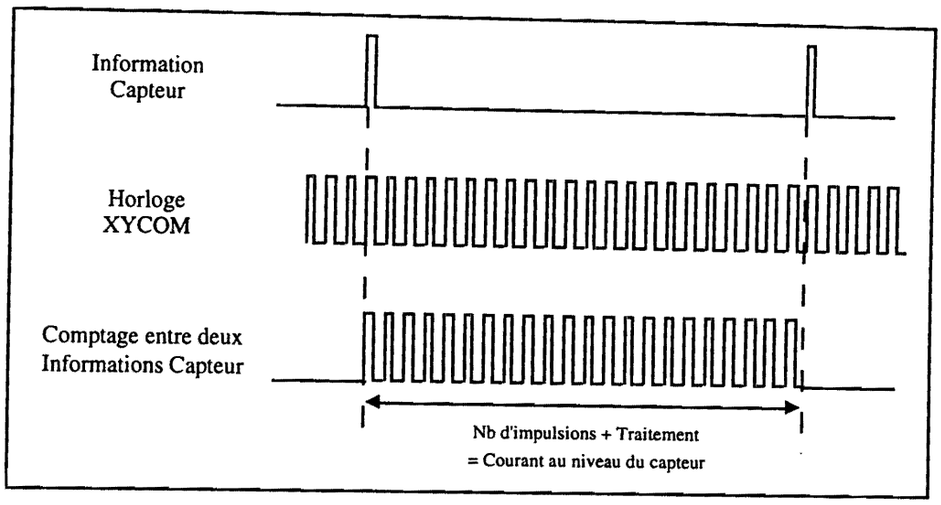


Figure B.23 - Utilisation de l'interface XYCOM

La figure B.23 montre le principe de fonctionnement d'une carte XYCOM. Pour un courant donné le capteur délivre des impulsions dont la fréquence lui est propre. La carte XYCOM mémorise dans un registre interne le nombre d'impulsions d'une horloge étalon entre deux signaux du capteur. Ce nombre d'impulsions associé à un coefficient permet de retrouver le courant. Pour éviter la mesure d'anciennes valeurs alors que les courants sont nuls, il a fallu prévoir dans la fonction d'acquisition une remise à zéro des registres spécifiques permettant de s'affranchir de cette contrainte.

4.2.3 Les automates

Les automates sont reliés, en règle générale, à l'un des ports séries de la carte CPU placée dans le châssis VME. La liaison est réalisée, soit par une liaison standard 3 fils pour les liaisons très courtes, soit par fibres pour les liaisons les plus longues.

4.2.3.1 Le TCS 1001

L'automate Balzers TCS 1001 [32] permet le contrôle d'une unité de pompage assurant le vide sur une certaine section du tube. Cette unité se compose principalement d'une pompe primaire, d'un pompe à vide élevé, d'une pompe ionique, d'une vanne à vide élevé et des cartes de mesure permettant de mesurer, ou de contrôler par des seuils, la pression à différents endroits de la machine (§1.3.3). Le domaine du vide mesuré s'étale du vide moyen (100 à 10^{-3} mBar) pour les jauges Pirani, jusqu'à l'ultra-vide en passant par le vide élevé (5.10^{-3} à 2.10^{-10} mBar) pour les jauges Penning. Toutes ces informations sont visualisées sur l'automate par un afficheur LCD et un panneau de voyants lumineux, ou sont remontées vers le *concentrateur* associé. Le principe de fonctionnement d'une unité de pompage est séquentiel. Lorsque le vide créé par la pompe à vide se trouve à une pression inférieure aux seuils des cartes de mesures, il est alors possible d'ouvrir la vanne à vide élevé qui créera le vide dans le tube (vide qui aura été mis auparavant à une pression inférieure à la pression atmosphérique). La pompe primaire est utilisée pour évacuer l'air que la pompe à vide élevée aura retiré du tube (§1.3.3). Les autres vannes sont des vannes de sécurité ou de régulation.

Cet automate est composé d'une carte de mesure Pirani, d'une carte de mesure à cathode froide pour

le vide élevé et l'ultra-vide, d'un module d'entrée, d'un module de sortie et d'une interface RS-232/485 pour le commander à distance.

4.2.3.2 Le TPG 300

Le contrôleur de pression Balzers TPG 300 [33] permet la mesure de la pression totale allant de la pression atmosphérique à 10^{-11} mbar dans des chambres à vide, des pompes et des installations d'expérimentation et de production. Grâce à des fonctions de commutation dépendant de la pression (relais actionnables en fonction de seuils de pression), il peut assurer certaines fonctions de commande et de surveillance pour équipements à vide. Dans notre cas, nous commandons des appareillages électriques extérieurs tels que les vannes. La précision de cet appareil dépend du type de jauges utilisées. Notre configuration est composée par une jauge de type Pirani et deux jauges de type Penning.

Le TPG 300 accepte jusqu'à 4 circuits de mesure simultanée (Pirani et/ou Penning) et dispose d'une carte d'interface comportant une interface RS-232-C et 5 relais de commutation. Trois types de cartes de mesure peuvent être utilisés :

- les cartes Pirani comprennent deux circuits de mesure de vide intermédiaire indépendants l'un de l'autre ;
- les cartes à cathode froide (Penning) comportent un circuit de mesure pour le vide élevé et l'ultra-vide ;
- les cartes mixtes comportent un circuit de mesure Pirani et un circuit de mesure Penning ;

Chaque carte de mesure dispose d'une sortie de signal analogique pour chaque circuit de mesure. Cette caractéristique est importante car la sortie analogique peut être reliée à une carte VADC pour mesurer la pression. Dans notre projet, nous utilisons jusqu'à présent cette configuration pour lire la pression à distance sans passer par un lien série. Le pilotage du TPG 300 à distance via un lien série est prévu, mais il ne sera pas mis en place afin d'économiser des liens série qui sont peu nombreux et, de ce fait, très prisés.

4.2.3.3 L'automate INCAA

Les automates INCAA [34] installés au niveau de l'injecteur et de l'aimant d'analyse permettent de commander les deux électro-aimants de déviation du faisceau afin de ne garder que les faisceaux de masses utiles. Chaque alimentation comprend six parties fonctionnelles :

- filtre sur le réseau de puissance ;
- puissance ;
- régulation ;
- protection et sécurité ;
- contrôle et commande locale ;
- contrôle et commande à distance.

4.2.3.4 Le teslamètre

Le teslamètre [35] est utilisé en collaboration avec la cible INCAA. Il mesure l'induction magnétique courante, soit au niveau de l'aimant d'analyse, soit au niveau de l'injecteur.

4.3 L'architecture logicielle

Des choix sur le plan logiciel ont été adoptés pour inaugurer la phase d'exploitation de l'accélérateur Vivitron. Ils doivent faciliter le développement d'un système informatique de contrôle et commande convivial et efficace. Les outils développés doivent pouvoir être adaptés au contrôle des équipements scientifiques associés au Vivitron, tels que les multidétecteurs (EUROGAM, DEMON, ICARE, EUROBALL...). De plus, le système informatique conçu doit être fortement orienté objet et doit déboucher sur une architecture distribuée. Son rôle sera multiple :

- contrôler les cartes d'interface et les automates ;
- gérer les protocoles de communication ;
- présenter les informations sous forme graphique ;
- donner une interactivité humaine avec l'utilisation de la souris pour les commandes ;
- gérer les alarmes mineures et les sécurités ;
- garder un historique des commandes et des mesures ;
- mettre en place des sauvegardes ;
- fonctionnement en mode dégradé.

Les logiciels choisis se composent d'un noyau temps réel dans les *concentrateurs*, d'un générateur d'interfaces graphiques animées et d'un système de gestion de base de données, le tout utilisé sur des stations de travail intégrant le système UNIX.

4.3.1 Le noyau Temps Réel

Le choix de l'exécutif temps réel VxWorks (§3.1) a déjà été fait pour les applications du multidétecteur EUROGAM où il donne entière satisfaction. Il sera utilisé pour la gestion des tâches, des mécanismes d'interruption et de sémaphores, pour les communications entres tâches locales ou distantes et pour la gestion des Entrées/Sorties. Par ailleurs, VxWorks s'intègre pleinement dans l'environnement de développement UNIX et intègre les réseaux pour la communication entre tâches distantes.

4.3.2 Le générateur d'interfaces graphiques animées

Nous avons choisi un outil graphique qui s'intègre dans l'ensemble et qui permet d'augmenter la productivité des développeurs d'Interfaces Homme-Machine (IHM) sur des stations de travail. Le produit choisi, SL-GMS (§3.4), est basé sur une méthodologie orientée objet et s'intègre pleinement dans l'e système X-Window. Il se compose essentiellement :

- d'un éditeur graphique interactif ;
- d'une interface entre les variables de l'application et les objets graphiques permettant ainsi l'indépendance entre code utilisateur et interface graphique ;
- d'un gestionnaire d'écrans permettant le prototypage directement pendant la phase de création des écrans graphiques.

Ce générateur sera utilisé pour créer les écrans graphiques de contrôle implantés sur les stations de

travail. Ces écrans seront manipulés par les opérateurs, à l'aide de la souris, pour contrôler et commander le Vivitron.

4.3.3 La base de données

Nous avons souhaité que cette base de données s'intègre bien dans un environnement orienté objet notamment avec celui nécessaire pour créer l'interface graphique. Pour cela, nous avons choisi le produit O_2 (§3.3) qui est fondé sur trois principes essentiels :

- l'application de l'approche orientée objet au monde des bases de données ;
- la combinaison des fonctionnalités offertes par un système de gestion de base de données, un langage de programmation orienté objet et une interface homme/machine ;
- le respect des standards du marché (UNIX, X-Window, Motif).

La base de données prend en charge, en plus de l'historique et de l'aspect matériel (*concentrateur*), l'aspect fonctionnel du système (fonctions de mesure et de commande, calibration, comportement graphique des écrans GMS, etc.) et la gestion des écrans GMS.

4.3.4 Conclusion

La mise en place d'une **architecture logicielle distribuée** sera possible grâce à l'utilisation de mécanismes de communication de l'environnement UNIX/VxWorks (les procédures à distance et le protocole X).

L'utilisation des IHM distribuées dans les systèmes embarqués devient de plus en plus courante. Les utilisateurs ne se contentent plus de commandes textuelles, ils veulent disposer d'un système multifenêtrage et interactif qui rendrait leurs applications plus faciles à utiliser. L'interface graphique la plus utilisée dans les systèmes embarqués est le standard X-Window [36]. Cette interface s'intègre pleinement dans les trois logiciels choisis, ce qui nous permet de **découpler l'application en processus clients et serveurs**. Ainsi, le travail peut être effectué sur chaque *concentrateur* et les résultats peuvent être affichés sur une station de travail unique.

Le système de gestion de bases de données O_2 va être utilisé pour intégrer dans la base, en plus de la description des équipements, le **code** des programmes devant les gérer. Cette fonctionnalité n'est pas habituelle dans un système de gestion de base de données qui a surtout pour rôle de décrire, stocker, accéder à de grosses quantités de données et les maintenir, d'où son originalité. La base va devenir ainsi un **élément indispensable** du système informatique de contrôle et commande.

— C —

Conception et Réalisation du système informatique

1 Les principes de conception du premier système

Le premier système informatique de contrôle et commande a permis, pendant 2 ans, un contrôle effectif et efficace du Vivitron tant pour la mise au point du générateur de tension que pour son exploitation. Notre travail a consisté à faire évoluer cette version vers un système plus souple et plus performant, doté de plus de fonctionnalités. Pour mieux comprendre l'évolution de ce système, il est bon de rappeler quelques caractéristiques de sa première architecture.

1.1 Les mesures

Les contraintes d'isolation galvanique (chapitre A, §1.4.2) nous ont obligés à utiliser des cartes du marché travaillant dans des conditions très sévères. Celles que nous avons trouvées ne possèdent pas les fonctionnalités d'interruption adaptées au traitement des différents paramètres du Vivitron. En effet, nous aurions voulu des cartes qui n'émettent une interruption que si la valeur mesurée dépasse un seuil de variation réglable. Ce choix nous a donc imposé de travailler en **scrutation**. Une grande boucle d'acquisition scrute les différents types de cartes les unes après les autres : cette fonction, assurée par la tâche **tPollAcq** (*Task Polling Acquisition*), effectue toutes les mesures au même rythme. La tâche *tPollAcq* accède aux données d'une mesure en faisant appel aux fonctions d'acquisition spécifiques à chaque type de carte. Un avantage de ce principe est de n'appeler les fonctions d'acquisition qu'une seule fois par boucle de scrutation au lieu d'une fois par mesure. Certains types de cartes, comme les VADC (chapitre B, §4.2.2.2) qui travaillent en mode recouvrement (un nouveau canal peut être sélectionné alors qu'une conversion est en cours) se prêtent parfaitement à cette méthode d'acquisition.

Dans notre première architecture, nous avions défini une **mesure** comme étant le **résultat d'une lecture sur un capteur**. Or, l'information transmise à l'opérateur pouvait provenir aussi bien du résultat d'une mesure que du résultat d'un calcul établi à partir de plusieurs mesures. Pour obtenir la grandeur utilisée au niveau des écrans de contrôle nous avions introduit la notion de **mesure calculée** (association de plusieurs mesures). La *mesure calculée* ne pouvait pas être intégrée dans la boucle d'acquisition car elle n'avait pas une affectation physique telle qu'un numéro de canal sur une carte d'acquisition. Son acquisition devait être gérée périodiquement par une tâche indépendante, ce qui nuisait à l'homogénéité du système et compliquait l'installation des mesures calculées.

Une mesure ou une *mesure calculée* était identifiée logiquement dans notre système par un nom symbolique sous la forme :

[référence équipement]_[référence mesure]_M
exemple : "AimantSW_PowerElectronicOn_M".

Dans chaque concentrateur, ce nom symbolique figurait sur une **table de symboles** propre aux mesures et indépendante de la table de symboles standard de VxWorks (chapitre B, §3.1.3). Les mesures étaient ainsi isolées des autres symboles et l'accès au symbole d'une mesure ne nécessitait pas la consultation de toute la table de symboles standard.

La figure C.1 nous montre le traitement des mesures dans ce premier système. Les mesures, gérées par la boucle d'acquisition, étaient calibrées et stockées dans une base de données temps réel. Elles utilisaient les RPC pour être remontées vers les écrans de contrôle. L'acquisition des mesures pour un automate diffèrent légèrement de celle des cartes d'interface VME. Comme les informations transitent sur une liaison série nous utilisions une boucle d'Entrée/Sortie qui faisait appel aux gestionnaires standard (*ioctl(), putc(), getc(), ...*). Cette boucle, exécutée sous forme de tâche, traitait les chaînes de caractères et mettait à jour un tableau qui regroupait les informations par canal. La fonction d'acquisition venait lire ce tableau pour faire son acquisition. Le problème qui se posait avec cette méthode était que la fonction d'acquisition pouvait remettre à jour la base de données temps réel plus rapidement que la boucle d'Entrée/Sortie ne remettait à jour le tableau.

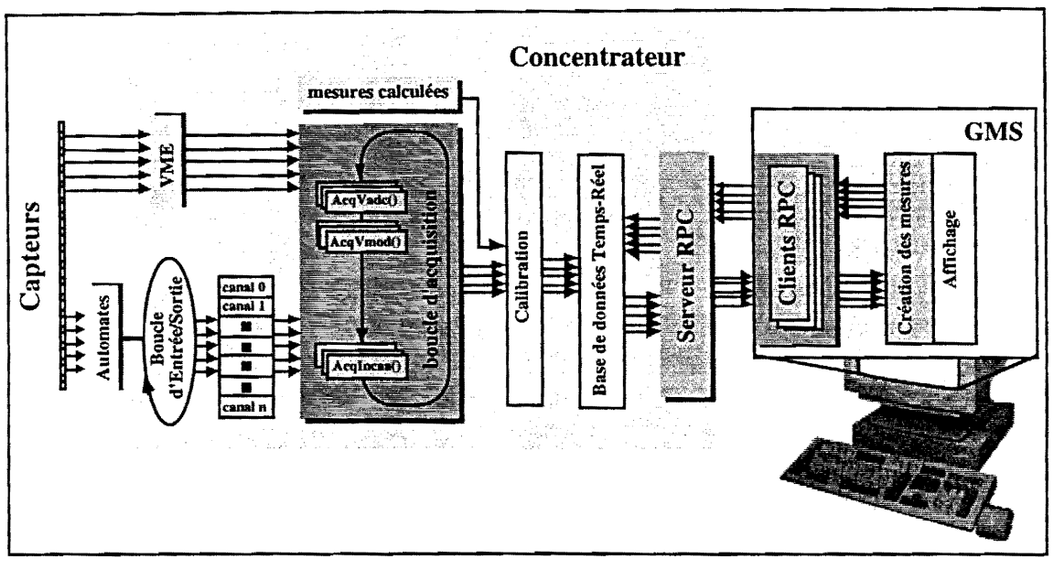


Figure C.1 - Le traitement des mesures

Pour des raisons propres à GMS nous avions installé le serveur RPC de mesures au niveau du *concentrateur* et les clients RPC de mesures au niveau de l'interface graphique, ce qui n'était pas sans conséquence pour le fonctionnement du système.

1.2 L'utilisation de GMS

Toute l'interface graphique a été réalisée avec le générateur d'interfaces graphiques animées GMS. Nous avons vue dans le paragraphe 3.4.3 du chapitre B la grande particularité de ce logiciel. Une application GMS est construite autour d'une boucle principale (figure C.2) dont la fonction essentielle est de se mettre en attente d'événements envoyés par le serveur X. Deux types d'événements nous intéressent tout particulièrement : les événements concernant la souris et les événements de type **gmstimer**. Nous allons utiliser le premier type d'événements pour déclencher les commandes, ce qui s'intègre pleinement dans l'architecture de GMS. Les traitements liés à notre propre application seront insérés dans la séquence de traitement attachée aux événements de type *gmstimer*.

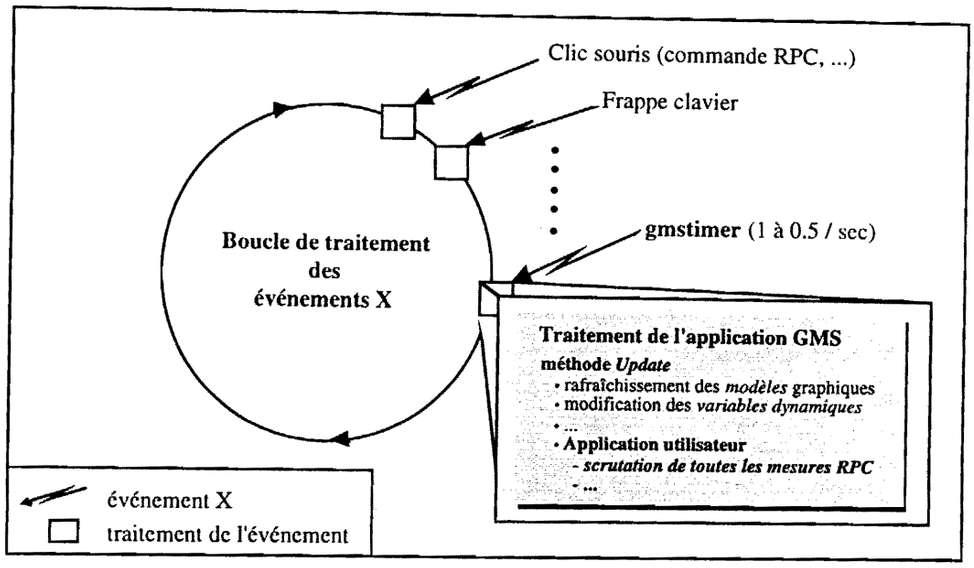


Figure C.2 - La boucle GMS

Ainsi GMS prenait en charge lui-même le traitement des mesures et nous contraignait à une scrutation périodique au rythme de *gmstimer*. Avec ce choix, il n'y avait pas de synchronisation entre le traitement de l'information (GMS) et la génération des données (concentrateur). Nous avons effectué des calculs de communication par RPC qui montrent qu'un tableau de 16 mesures d'un même paramètre peuvent être remontées depuis les *concentrateurs* vers les écrans GMS toutes les 10 millisecondes, ce qui nous donne une scrutation de *160 mesures/seconde* sur un paramètre. Or, la période d'échantillonnage, qui est donnée par la période de l'appel de la méthode *update* de GMS (0,5 s à 1 s), est de l'ordre de la **demi-seconde**. Il est clair que GMS imposait une fréquence d'échantillonnage (2 mesures/seconde) qui n'était pas du tout adaptée à la scrutation (160 mesures/seconde) ce qui ne nous permettait pas d'exploiter au maximum les *concentrateurs*. De plus, le fonctionnement de GMS ne donnait pas la possibilité au *concentrateur* d'interpeller un écran GMS, ce qui limitait les possibilités de traitement de situations exceptionnelles (claquages...).

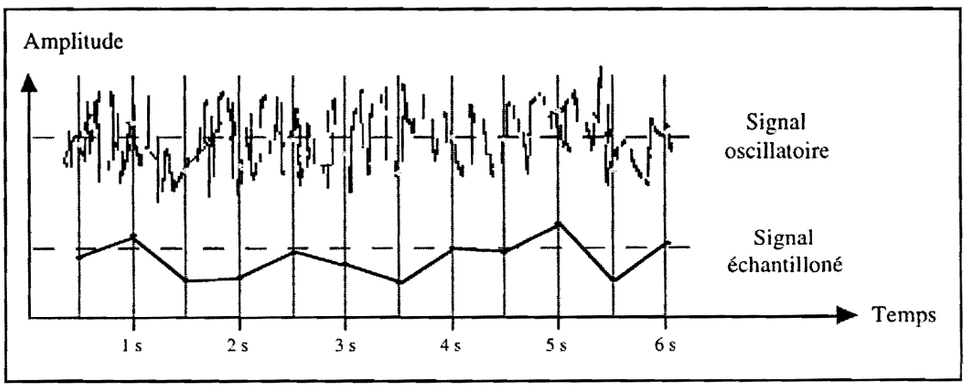


Figure C.3 - Echantillonnage d'un signal oscillatoire

Nous savons que le Vivitron est une machine relativement lente (chapitre A, §2.2) et que les paramètres, une fois la machine stabilisée et dans des conditions de fonctionnement normales, peuvent

maintenir une valeur constante pendant quelques minutes. Dans ce cas de figure, la fréquence d'échantillonnage imposée par GMS s'avère suffisante. Cependant, dans le traitement de phénomènes rapides comme par exemple la stabilisation en tension par effet Corona qui a un temps de réponse approximatif de 80 ms (chapitre A, §2.2), cette fréquence s'avère largement insuffisante et aboutit à une dégradation de l'information. Prenons comme exemple le signal oscillatoire bruité de la figure C.3. Après échantillonnage le signal obtenu n'a aucune ressemblance au signal réel.

1.3 Les commandes

Comme pour les mesures, nous avions défini une **commande** comme étant le **résultat d'une action sur un actionneur**. De la même façon, nous avions introduit la notion de **commande calculée** (association de plusieurs commandes) pour obtenir la commande utilisée au niveau des écrans de contrôle. En revanche, une *commande calculée* était traitée au niveau du *concentrateur* comme une commande standard : un clic de la souris, au niveau des écrans de contrôle, déclenche un client RPC qui réveillera le serveur RPC de commandes au niveau du *concentrateur* (figure C.4). Pour ne pas bloquer l'écran de contrôle, il faut que le serveur RPC réponde rapidement, donc qu'il soit libre en permanence. C'est ce qui justifie l'utilisation d'une tâche pour exécuter la commande, dès que celle-ci est lancée le serveur est libéré. Pour éviter que deux commandes s'exécutent en même temps, le serveur refuse de lancer une nouvelle tâche tant que la précédente n'est pas terminée. Dans ce cas, le serveur RPC envoyait au client un état signifiant qu'il était occupé.

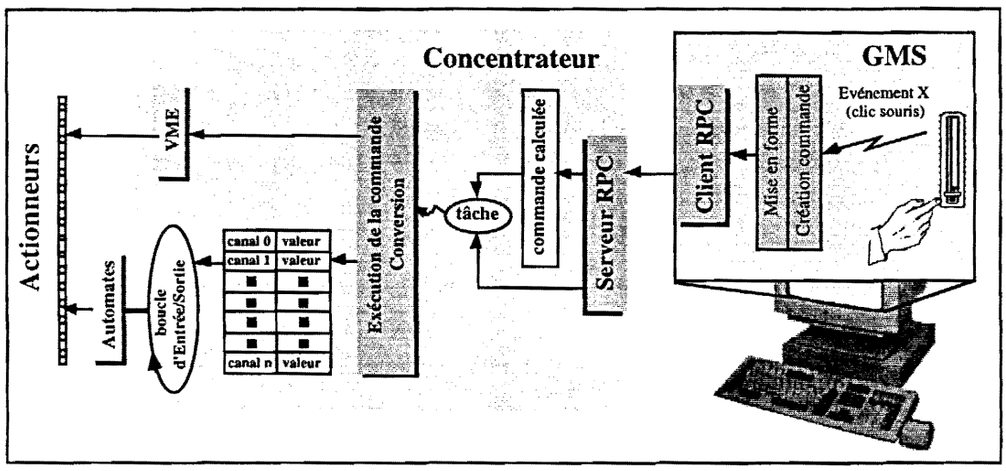


Figure C.4 - Le traitement des commandes

L'exécution d'une commande adressée à un automate utilisait le même principe retenu dans l'acquisition des mesures (figure C.4). Nous utilisions une boucle d'Entrée/Sortie lancée sous forme de tâche et faisant appel aux gestionnaires standard. La tâche de commande allait déposer la valeur de la commande sur un tableau de canaux. La boucle d'Entrée/Sortie scrutait régulièrement ce tableau et, si elle détectait une valeur modifiée, elle émettait alors la commande vers l'automate. L'inconvénient de cette méthode se traduisait par le fait que la commande n'était pas prioritaire par rapport aux mesures. Elle devait attendre que la boucle ait effectué toutes les mesures, et ce n'était qu'à la fin du traitement que la commande était envoyée à l'automate.

Une commande ou une *commande calculée* était identifiée logiquement dans notre système par un nom symbolique sous la forme :

[référence équipement]_[référence commande]_C
exemple : "AimantSW_PowerElectronicOn_C".

Dans chaque concentrateur, ce nom symbolique figurait sur une table de symboles propre aux commandes et indépendante de la table de symboles standard de VxWorks (chapitre B, §3.1.3).

Dans l'environnement du Vivitron, certaines commandes se composent d'impulsions à durée variable et peuvent utiliser plusieurs canaux. C'est le cas de la commande marche/arrêt (**On/Off**) du moteur d'entraînement de la courroie. Pour la mise en marche du moteur, GMS générait une impulsion et envoyait une commande **On** au concentrateur. Pour arrêter le moteur, il générait une deuxième impulsion et envoyait une commande **Off**. GMS fonctionne au rythme de *gmstimer*, lui-même fortement dépendant de la charge de la machine hôte : nombre de mesures, charge du serveur X. Le serveur X restait bloqué tant que le bouton gauche de la souris restait enfoncé ce qui bloquait les écrans et faisait durer la commande en cours. Il en résultait une absence de déterminisme temporel des écrans de contrôle, à moins de les inhiber le temps d'une commande. En conclusion, il n'était pas optimum de demander à GMS de fournir des signaux de commandes complexes.

Nous venons de voir qu'une application GMS est construite autour d'une boucle qui gère les événements X. Il faut savoir qu'un seul événement est traité à la fois. Par conséquent, si le traitement d'un événement est bloqué, pour un motif quelconque (charge processeur trop importante sur les mesures, *timeout* RPC...), l'application GMS peut rester momentanément bloquée à son tour. Cette situation pouvait être dangereuse lors de l'exécution d'une commande. L'opérateur qui cliquait sur un bouton de l'écran graphique pour passer une commande, pouvait être surpris s'il ne voyait pas réagir le bouton instantanément. Il croyait alors que la commande n'était pas passée et il insistait à nouveau au risque de passer plusieurs fois la même commande. Toutes ces commandes s'exécutaient en rafale quand l'écran se débloquait. Cette situation pouvait arriver quand le serveur de mesures restait bloqué. Les clients de mesures insérés dans la boucle de traitement de l'événement *gmstimer* se bloquaient à leur tour et nous nous retrouvions dans la situation décrite.

Nous illustrons ce phénomène particulier avec la commande de démarrage du moteur d'entraînement de la courroie. Cette commande appartient à l'écran de contrôle qui est associé, entre autres, au *concentrateur* du terminal. Ce *concentrateur* est alimenté par un alternateur entraîné par la courroie et donc, tant que la courroie ne tourne pas, il n'est pas sous tension. Dans notre premier système, cette situation ne permettait pas de lancer le serveur de mesures. Pendant l'arrêt de la courroie, l'écran de contrôle demandait des mesures au serveur de mesures du terminal qui ne répondait pas et allait bloquer les clients de mesures, ainsi que le traitement de l'application GMS (figure C.2). Quand l'opérateur cliquait sur le bouton de démarrage du moteur, la commande était lancée par événement X, mais le bouton ne réagissait pas car l'application GMS était bloquée. Une fois l'électronique du *terminal* mise sous tension, le serveur de mesures était lancé et les clients RPC se débloquaient ainsi que l'application GMS. Ce phénomène pouvait durer une vingtaine de secondes, temps pendant lequel l'opérateur ne savait pas si la commande était passée ou non, sauf par un écran vidéo.

Pour les commandes associées à des asservissements, nous avions un traitement particulier. Prenons comme exemple la commande d'une vanne. Celle-ci se déroule en trois étapes :

- mise en route du moteur de déplacement de la vanne ;
- lecture de la position de la vanne jusqu'à atteindre la position donnée par l'opérateur ;
- arrêt du moteur.

Ces trois actions étaient gérées par une tâche indépendante créée uniquement pour la commande des vannes. Le fichier de démarrage du concentrateur, le *startup*, devait lancer autant de tâches qu'il y avait d'asservissements. Ces tâches étaient réveillées par la tâche de commande.

1.4 Le démarrage du système

Dans la procédure d'initialisation, nous faisions appel à la base de données O_2. Cependant, son rôle se limitait à stocker une brève description des paramètres concernant les *concentrateurs* et non les écrans graphiques. L'ensemble des paramètres provenaient donc de deux origines différentes : O_2 et GMS.

1.4.1 Les écrans

Lors de l'initialisation de l'interface graphique, GMS ne faisait pas appel à la base O_2. Les informations comme la plage de variation des différentes mesures affichées, les seuils d'alarme et d'incidence, les aspects graphiques, les coefficients de calibration pour les commandes... étaient prises en charge directement par GMS. Cette attitude ne garantissait pas la cohérence entre les paramètres pris en charge par les écrans GMS et les paramètres utilisés par les *concentrateurs* qui étaient, quant à eux, répertoriés dans la base.

1.4.2 Les *concentrateurs*

Lors du démarrage des *concentrateurs* ou d'une reprise sur incident, chaque *concentrateur* demandait à la base la description des paramètres qu'il devait gérer. En fonction de ces données il s'auto-configurait. Cette étape était assurée par une procédure de configuration appelée *conclnit*. A l'origine, la communication entre les *concentrateurs* et O_2 s'appuyait sur les *sockets* par l'intermédiaire d'un client spécifique de O_2 et d'un client VxWorks (figure C.5).

Cette méthode nous obligeait à être dépendant de la machine hôte contenant le serveur O_2. En effet, ce dernier n'a qu'une licence dédiée à une machine unique et le transfert sur une autre machine n'est donc pas autorisé. Un dysfonctionnement de la machine hôte bloquerait tout le système. Pour pallier cet inconvénient et améliorer les temps de chargement des paramètres, ceux-ci étaient chargés à partir de fichiers de description des paramètres : les fichiers *o2resume*. La tâche *conclnit* interprétait son fichier de description avant de configurer le concentrateur. L'administrateur du système pouvait, à tout moment, connaître le contenu de la base de données, soit en la consultant, soit en lisant les fichiers *o2resume* qui regroupaient les mesures et les commandes par concentrateur.

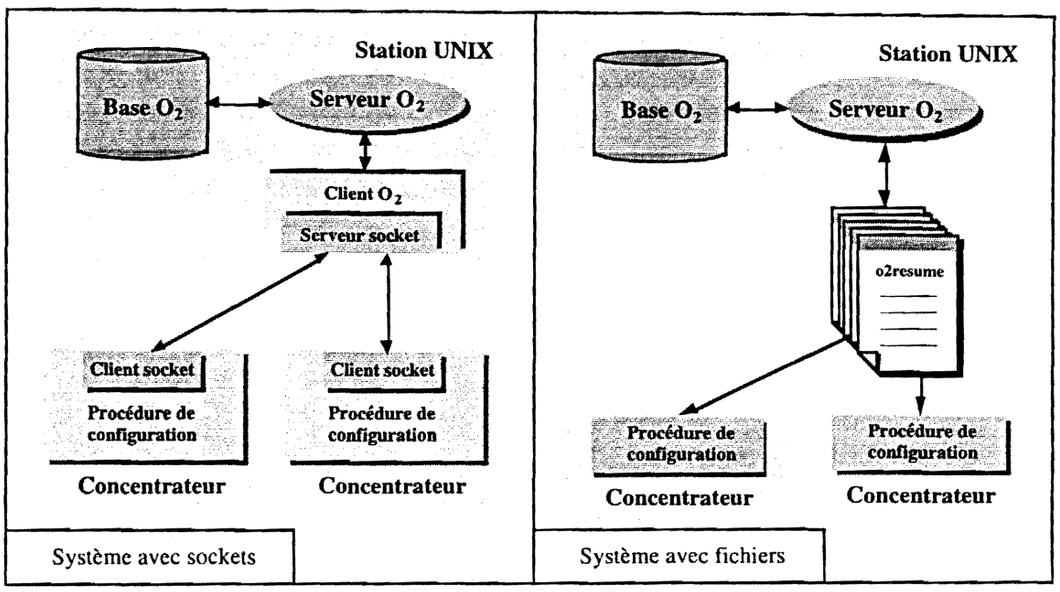


Figure C.5 - Communication O_2 - concentrateurs, au démarrage

2 Les concepts de base du système actuel

Le premier système informatique de contrôle et commande permettait, nous le rappelons, un contrôle effectif et efficace du Vivitron. Aucune modification ne sera apportée à son architecture matérielle. Cependant aux cours de son exploitation nous avons pu constater certaines limites dans son architecture logicielle :

- saturation sur les affichages graphiques des stations de travail ;
- mauvaise perception de l'évolution temporelle des mesures ;
- l'état réel du système n'apparaît pas sur les écrans de contrôle ;
- la cohérence entre les modules n'est pas garantie.

Pour y remédier nous avons fait certains choix :

- utiliser au mieux la puissance de calcul ;
- rendre les *concentrateurs* autonomes ;
- reconsidérer le mode de communication entre le monde UNIX et les *concentrateurs* ;
- élargir l'usage de la base de données O_2 afin qu'elle garantisse la cohérence du système ;
- établir des règles de programmation pour présenter une structure plus limpide et améliorer ainsi la maintenance.

2.1 Utilisation de la puissance de calcul

La puissance de calcul dans notre système de contrôle et commande est surtout concentrée dans les *concentrateurs* pour le traitement des données. Chaque *concentrateur* a une puissance de calcul importante : **20 MIPS** (Million d'Instructions Par Seconde) pour un **68040** à architecture SISC (*Complexed Instruction Set Computer*) et **3,5 MFLOPS**. Pour ne pas consommer inutilement cette puissance, nous avons décidé de ne transférer que des données pertinentes.

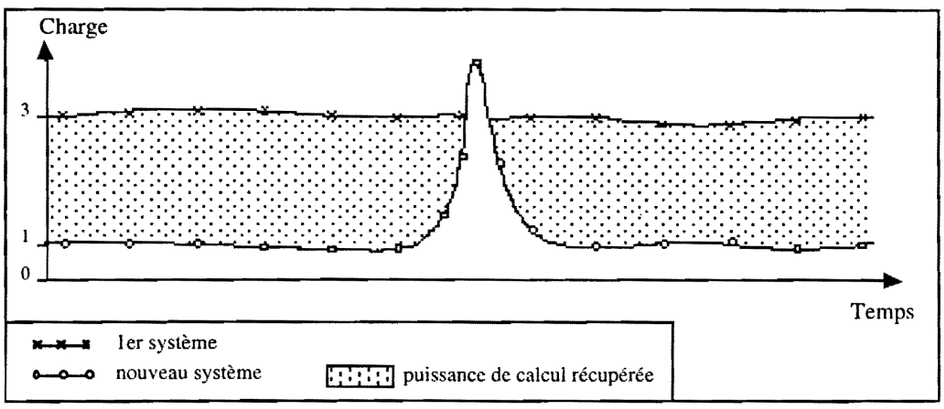


Figure C.6 - Puissance de calcul des concentrateurs

La solution proposée a été de traiter au plus près du capteur, **au niveau du concentrateur**, l'évolution d'une mesure (§3). Ainsi, celle-ci n'est remontée vers les écrans de contrôle que dans les trois cas de figure suivants :

- un changement sensible de sa valeur est constaté ;
- à la demande de l'opérateur (relecture d'une commande, par exemple) ;
- remontée périodique pour assurer un renouvellement minimum.

Cette solution apporte un gain de puissance de calcul important sur l'ensemble informatique, comme le montre la figure C.6. La charge dans le premier système était importante, mais relativement constante et surtout mal répartie. Dans notre système actuel, elle s'affaiblit considérablement, sauf, lors d'événements sporadiques (par exemple les claquages) où une remontée importante de mesures sera dirigée vers les écrans de contrôle.

2.2 L'autonomie des *concentrateurs*

L'utilisation de GMS dans notre premier système (§1.2) interdit à l'opérateur d'avoir une image sur l'état global du système à un instant donné, ce qui l'empêche de réagir instantanément en cas de panne. Le *concentrateur* ne peut pas gérer la remontée d'informations vers les écrans de contrôle ce qui entraîne la remontée de mesures inutiles et retarde la remontée de messages prioritaires. C'est GMS qui prend en charge la lecture des mesures.

L'idée est de rendre les *concentrateurs* **autonomes**. C'est-à-dire que les *concentrateurs* envoient des informations vers un écran GMS à leurs propres initiatives et tolèrent les pannes. Ainsi, les échanges entre les *concentrateurs* et les écrans sont réduits, et la puissance de calcul récupérée reste disponible pour avoir une meilleure dynamique des éléments de commande graphique. De plus, un mauvais comportement d'un *concentrateur* (perte de liaison, panne...) ne nuit en rien aux fonctions dépendant des autres *concentrateurs*. Chaque écran graphique pourra représenter l'état réel des *concentrateurs* auxquels il est attaché. Un serveur d'alarmes indique à l'opérateur un défaut du système par des messages sonores et une impression au fil de l'eau. Les messages d'alarmes ou d'erreur peuvent aussi être consultés dans la base de données.

2.3 Redéfinition des communications

Nous avons vu dans le paragraphe 1.2 que GMS nous contraint à une **scrutation périodique**. Les communications utilisées pour les mesures utilisent le support des procédures à distance (RPC). GMS devient ainsi un client RPC des serveurs RPC se trouvant dans chaque concentrateur. Ce sont les écrans GMS qui interrogent les *concentrateurs* ce qui rend ces derniers dépendants de GMS. Cette dépendance peut bloquer le système pendant quelques secondes ce qui entraîne un empilement des commandes (§1.3).

La solution est d'**inverser les rôles** pour que ce soient les *concentrateurs* qui interrogent les écrans GMS quand ils l'estiment opportun. L'idée est de ne plus utiliser les **RPC** comme support de communication entre les écrans et les *concentrateurs*, mais le support adopté par les applications GMS : les **événements X**. Des clients X sont lancés sur chaque *concentrateur* et envoient, par messages intégrés dans le flux d'événements X, toutes les informations nécessaires à l'affichage d'une mesure. La scrutation périodique est ainsi supprimée et GMS peut travailler en serveur, les *concentrateurs* étant ses clients.

Le travail est distribué à tous les *concentrateurs* clients et les résultats sont récupérés par l'application GMS serveur qui les affichera sur ses écrans. Toutes les autres communications inter-processus gardent le support de l'exécution de procédures à distance en mode TCP. La présentation des différents paragraphes de ce chapitre sera axée sur la communication par événements X car c'est la partie sur laquelle j'ai travaillé.

2.4 Une cohérence garantie par le SGBDOO O2

La dispersion des différents modules entraîne, lors de chaque modification, un risque réel d'introduire un défaut (symboles dupliqués...), car il devient très difficile de garantir la cohérence entre les modules.

O_2 nous permet de décrire la totalité du système sous forme d'un schéma d'objets. Ces performances nous ont permis d'asseoir la construction de notre nouvelle architecture autour de cette **base** qui devient, ainsi, un **élément indispensable.** Son rôle ne se limite plus à stocker une brève description sur les paramètres que chaque *concentrateur* doit lire lors du démarrage du système. Elle contient la description physique et fonctionnelle de chaque *concentrateur* et le comportement graphique de chaque écran GMS. Nous avons voulu intégrer dans la base, en plus de la description du système de contrôle et commande, le code des programmes devant gérer ce système. L'unicité des données et des versions opérationnelles est ainsi garantie et l'introduction d'une éventuelle erreur lors d'une modification disparaît. De plus, pour éviter une fastidieuse réécriture du même code pour chaque nouveau paramètre, nous avons réalisé un outil qui permet au SGBDOO O_2 de générer ce code. Cette originalité autorise une modification aisée des fonctions existantes et évite de passer par un spécialiste. Le système devient ainsi **plus modulaire** avec une **structure plus limpide**, et la **maintenance est facilitée.**

Toutes les mesures et commandes effectuées par les *concentrateurs* sont datées et enregistrées dans O_2 et peuvent être consultées, à l'aide de l'historique, sur une interface graphique conviviale. Les chapitres 7 et 11 développent plus profondément le rôle de la base O_2 en insistant sur le travail que j'ai développé.

2.5 Les règles de programmation

Nous avons voulu donner une orientation objet à notre système informatique de contrôle et commande, pour profiter des avantages de la technologie objet : **maintenance, réutilisation** du code et **portabilité** du système, facilités (chapitre B, §3.2). Toute la modélisation objet de notre système se trouve dans le schéma d'objets de la base de données O_2. C'est pourquoi, l'utilisation d'un langage de programmation orienté objet tel que C++ ne se justifie pas vraiment. Nous avons donc conservé le langage C ANSI en y appliquant, toutefois, une philosophie de programmation par objets.

Pour pouvoir réutiliser nos modules informatiques nous pourrions créer une bibliothèque de fonctions. Mais cette solution ne nous permet pas d'atteindre la **réutilisabilité** souhaitée parce qu'une fonction,

simple unité de traitement, dépend fortement de son environnement, en particulier des structures de données qu'elle manipule. Pour maximiser la réutilisabilité, il est nécessaire de s'affranchir au maximum de la représentation de données. Cette méthode nous conduit à définir un ensemble de règles de programmation :

- Associer les fonctions aux structure de données qu'elles manipulent.
- Regrouper dans un même module les fonctions manipulant les mêmes structures de données. Nous obtenons ainsi une **encapsulation** des données et des traitements associés et nous pouvons alors assimiler un **module** à ce que la méthodologie orientée objet appelle **classe** (chapitre B, §3.2).
- Cacher les structures de données : ce sera l'implémentation du module. Nous interdirons à l'utilisateur de les manipuler les données directement. Nous pourrions associer cette **implémentation** à l'**instanciation** d'une classe dans la programmation objet.
- Offrir à l'utilisateur un ensemble de **services** (fonctions) manipulant ces structures de données. Les services sont l'équivalent des **méthodes** dans la programmation objet.

L'*abstraction des données*, résultante des deux dernières règles, assure que l'utilisation du module est bien dépendante de la façon dont sont organisées les données.

Le découpage de notre application se fera alors naturellement en modules correspondant à des parties de l'application fonctionnellement indépendantes. Chaque module contiendra donc un ensemble de structures de données et un ensemble de fonctions les manipulant, et sera présenté par :

- un **fichier source** contenant l'implémentation du module (définition des fonctions et des constantes, types et macros à usage interne) ;
- un **fichier en-tête** contenant les services offerts à l'utilisateur du module (les prototypes des fonctions externes et les définitions des constantes, types et macros).

2.6 La notion de paramètre et la notion d'ordre

Au cours de l'exploitation du système de contrôle et commande nous avons pu mieux cerner les profondes différences d'interprétation sur le sens à donner aux concepts de mesure et de commande selon que l'on se place du point de vue de l'opérateur ou du point de vue de l'électronicien. Pour l'opérateur une mesure renseigne sur un aspect de l'état de la machine et une commande est une action ayant pour objet de modifier cet aspect. Tandis que pour l'électronicien, une mesure est le résultat d'une lecture sur un capteur et une commande est une action réalisée par un actionneur. Pour clarifier le débat nous avons défini un vocabulaire commun à tous :

- l'opérateur effectue des **mesures** et exécute des **commande** ;
- l'électronicien effectue des **lectures** et exécute des **ordres**.

2.6.1 Définition d'une mesure : suite de lectures

La figure C.7a nous montre les différences entre la première architecture où l'on se plaçait du point de vue de l'électronicien et le système actuel où l'on se place du point de vue de l'opérateur :

- premier système : 1 mesure = 1 **lecture** ;
- deuxième système : 1 mesure = 1 **suite de lectures.**

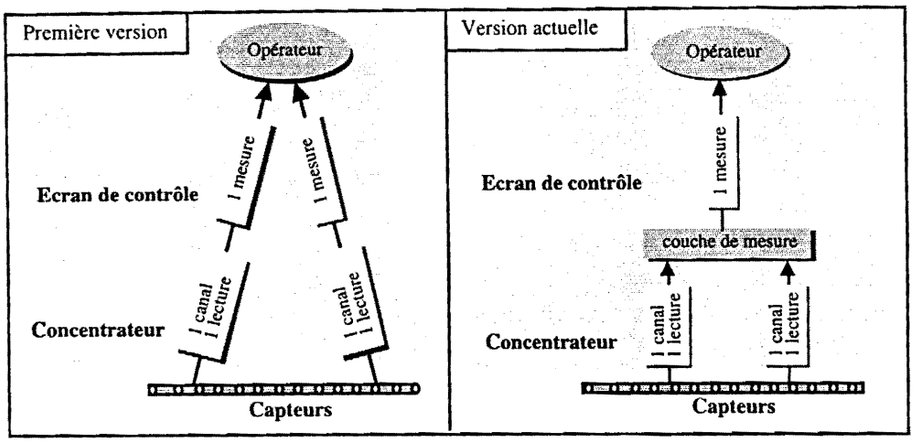


Figure C.7 a - Les mesures

A partir du moment où une mesure n'est plus liée à un capteur unique mais à un ensemble de capteurs, la notion de *mesure calculée* n'a plus de raison d'être. C'est une nouvelle couche qui transformera la mesure en une **suite de lectures** et qui mettra en place les fonctionnalités de la mesure. Chaque mesure sera associée à une fonction personnalisée (§3.3) créée dans cette nouvelle couche et faire une mesure reviendra à exécuter cette fonction. La tâche de gestion des *mesures calculées* disparaîtra et toutes les mesures seront alors insérées dans la boucle d'acquisition.

2.6.2 Définition d'une commande : suite d'ordres

Par analogie aux mesures les commandes seront définies de la façon suivante :

- premier système : 1 commande = 1 **ordre** ;
- deuxième système : 1 commande = 1 **suite d'ordres.**

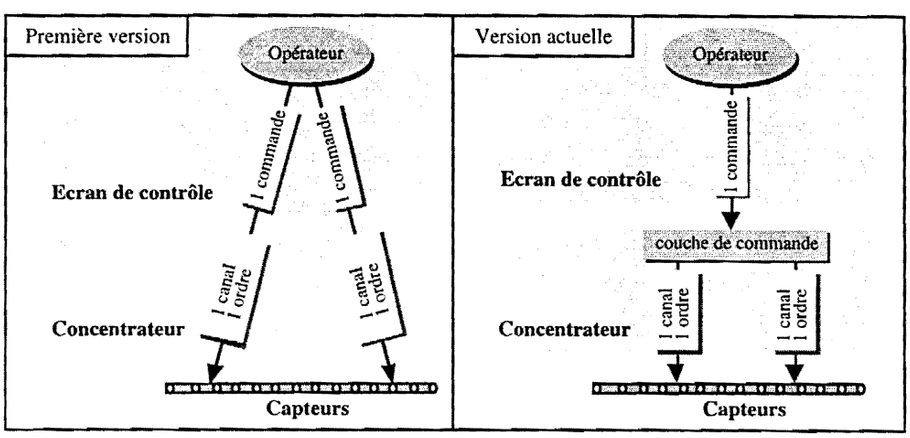


Figure C.7 b - Les commandes

Une commande n'est plus liée à un actionneur unique mais à un ensemble d'actionneurs et la notion de *commande calculée* disparaît. Les commandes bénéficieront, comme pour les mesures, d'une nouvelle couche qui transformera une commande en une **suite d'ordres** (figure C.7b). Chaque commande sera associée à une fonction personnalisée (§4) créée dans cette nouvelle couche.

2.6.3 Définition d'un paramètre : liste d'ordres

Nous venons de voir les concepts d'*ordre* et de *lecture*. Tous les deux représentent, la même chose : un accès à un canal d'une carte d'interface. Nous pouvons donc fondre ces deux concepts dans celui d'*ordre*, en sachant qu'un *ordre* peut être lu ou écrit. Les mesures et les commandes correspondent, toutes les deux, à l'exécution d'une suite d'*ordres*. Nous fondrons ces deux concepts dans celui de **paramètre**. Faire une mesure ou une commande revient alors à exécuter la **fonction associée au paramètre**. Si nous récapitulons :

- *un paramètre est une suite d'ordres ;*
- *une mesure est un paramètre doté d'une suite d'ordres lus ;*
- *une commande est un paramètre doté d'une suite d'ordres écrits et d'une suite d'ordres lus.*

Dans notre premier système nous accédions aux mesures et aux commandes directement par leur symbole. Elles étaient référencées sur deux tables de symboles séparées et indépendantes afin d'éviter la consultation de la table de symboles standard de VxWorks (§3.1.3). Elles ne pouvaient pas être consultées sous le *shell* VxWorks. Le *paramètre*, lui, sera référencé dans la table de symboles standard. Il pourra être manipulé facilement à partir du *shell* VxWorks. Cela est très utile pour la mise au point du système mais représente un danger certain pour l'exploitation, dans le cas où quelqu'un ouvre une session sur le concentrateur.

Le *paramètre* est maintenant une variable globale accessible indirectement par son symbole ou directement par son adresse. Les accès rapides aux *paramètres* de type mesure, lors de l'acquisition, ne se font plus par leur symbole mais par une table de pointeurs spécifiquement créée au démarrage (§3.1). Les accès aux *paramètres* de type commande se font directement par leur adresse (§4). Ainsi, l'utilisation de la table de symboles standard n'est plus un handicap.

Le *paramètre* est modélisé, au sein du concentrateur, par une structure de données appelée ***Paramètre*** *et contenant* :

- *le symbole de la mesure ou de la commande ;*
- *un mot d'état pour les droits d'accès et un mot d'état pour les actions autorisées ;*
- *la consigne de la commande ;*
- *la mesure associée à la commande (relecture de la commande envoyée ou lecture d'une mesure associée à la commande) ;*
- *une fonction de traitement (ou fonction associée au paramètre) ;*
- *les données de calibration pour la mesure et les données de conversion pour la commande ;*
- *la valeur courante datée et son mot d'état ;*
- *la valeur précédente datée et son mot d'état ;*
- *les paramètres d'évaluation pour la remontée d'une mesure vers les écrans (variation minimale, période de remontée, paramétrage de mise en trace (§10) et de mise en base) ;*
- *une "trace" des dernières valeurs courantes datées.*

2.6.4 Les mots d'état d'un *paramètre*

Pour mieux contrôler le traitement d'un *paramètre* nous lui avons associé deux mots d'état. Le premier, accessible par le champ *accès* de la structure ***Paramètre***, contient les droits d'accès au *paramètre*. Le deuxième, accessible par le champ *interdictions*, contient les actions autorisées sur le *paramètre*. La figure C.8a présente la composition choisie pour les deux mots d'état.

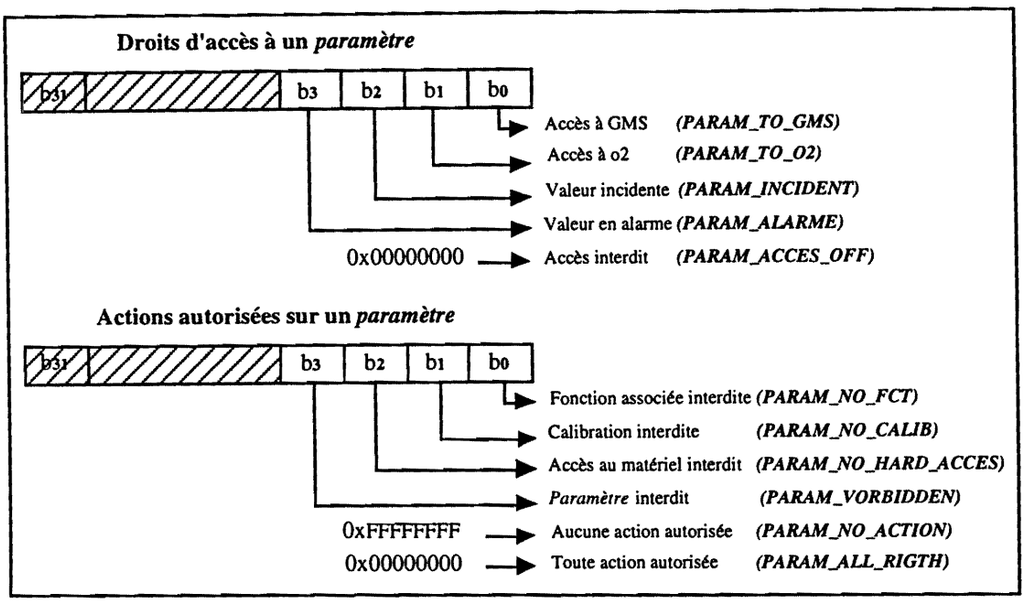


Figure C.8a - Les mots d'état d'un paramètre

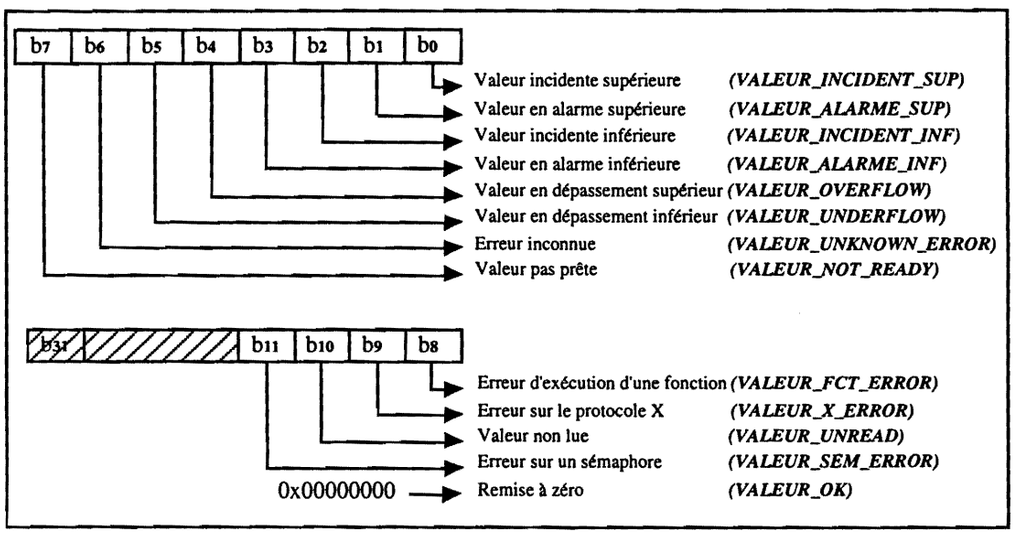


Figure C.8 b - Le mot d'état de la valeur d'un paramètre

Ces deux mots d'état sont utilisés par les différents services qui traitent les *paramètres*. Par exemple, la fonction *forceParamToGMS()* (§3.1) force la remontée des mesures vers les écrans GMS en activant le bit PARAM_TO_GMS. La fonction *mesureEcrans()* (§3.1) ne remonte les mesures vers les

écrans GMS que si le bit PARAM_TO_GMS est actif. Si le *paramètre* est en incidence ou en alarme, la fonction *parametreAccesOn()* (§3.1) active les bits PARAM_INCIDENT ou PARAM_ALARME.

Pour les actions autorisées sur un *paramètre*, prenons l'exemple de la fonction *acqCalcul()* (§3.2) : si le bit PARAM_NO_FCT est actif *acqCalcul()* n'exécute pas la fonction associée au *paramètre*. Prenons un dernier exemple : si le bit PARAM_NO_HARD_ACCES est actif le serveur de commandes du *concentrateur* (§4) n'exécute pas la commande.

Les **valeurs** prises par les paramètres ont, elles aussi, un mot d'état. Nous pouvons connaître, ainsi, avec plus de détails tout problème rencontré par la valeur d'un *paramètre*. Ce mot d'état est utilisé par les fonctions de calibration, de commande, de mesure, etc. Le premier octet de poids faible nous renseigne sur les erreurs de calcul (dépassement, alarme ou incidence) ainsi que sur la disposition de cette valeur. Le deuxième octet nous montre l'erreur induite sur la valeur suite à un problème de son environnement d'exécution (erreur de sémaphore lors du parcours de la liste des clients GMS (§6.2.2), erreur de protocole X lors de l'envoi de l'événement X, erreur lors de l'exécution d'une fonction qui utilise la valeur du *paramètre*). La figure C.8.b présente toutes les valeurs prises par le mot d'état associé à la valeur du *paramètre*. Ce mot d'état sera mis à jour par la *fonction de calcul* (§3.2).

2.7 Architecture générale

Après avoir décrit les concepts de base du système actuel, nous présentons, sur la figure C.9, les différents modules qui composent ce système. Dans les paragraphes qui vont suivre nous détaillerons plus particulièrement les communications X, l'acquisition, les commandes, le gestionnaire d'écrans (**xvivetat**), ainsi que les modules qui sont pris en charge par la génération de code dans le système de base de données O_2.

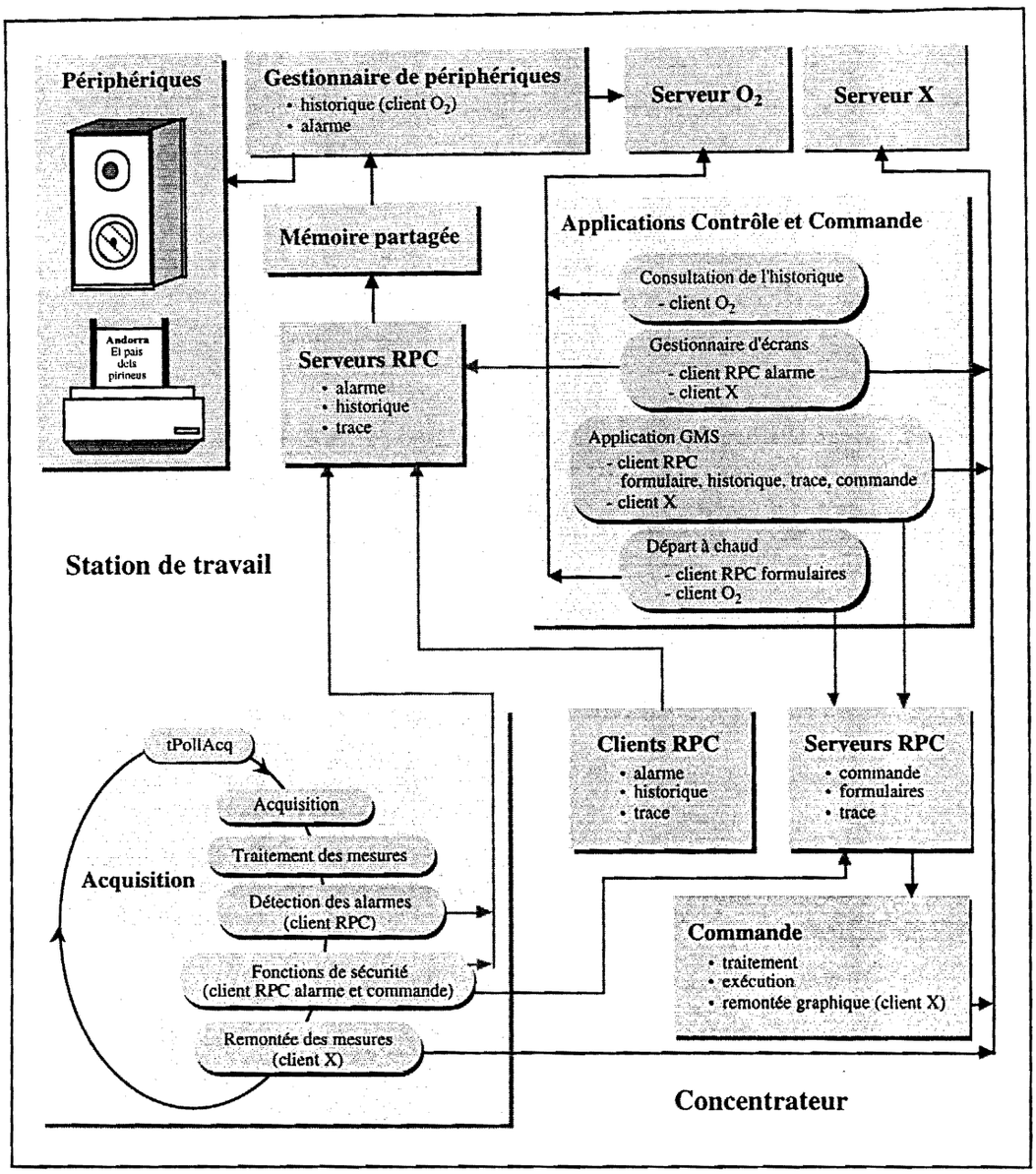


Figure C.9 - Architecture générale du système informatique de contrôle et commande

3 Traitement des mesures

Une mesure, depuis son acquisition sur les cartes d'interface jusqu'à son affichage sur les écrans graphiques, subit une série de traitements regroupés sur des modules spécifiques. Ces modules ont été créés pour bien séparer :

- le travail à effectuer par les écrans graphiques : l'affichage de la donnée ;
- le travail à effectuer par les *concentrateurs* : le traitement de la donnée jusqu'à son interprétation par l'opérateur ;
- le travail à effectuer au plus près du matériel : l'acquisition de la donnée brute.

Notre objectif est de **porter "l'intelligence" au plus près du matériel**, c'est-à-dire auprès des *concentrateurs*, et de **créer un système modulaire**. Le traitement des mesures, au niveau du concentrateur, doit conduire à la mise à jour d'une valeur directement exploitable par GMS.

3.1 La tâche d'acquisition : *tPollAcq*

La tâche *tPollAcq* conserve le même principe que dans la première version mais avec des fonctionnalités supplémentaires à savoir : la détection des alarmes, l'exécution des fonctions de sécurité et la remontée des mesures (figure C.10). A chaque tour de boucle, *tPollAcq* effectue les opérations suivantes :

- **Lie les valeurs physiques sur un capteur ou sur un automate**
  Les fonctions d'acquisition, regroupées par type de carte, sont exécutées les unes après les autres.
- **Exécute les fonctions des paramètres**
  La *fonction de calcul* (§3.2) exécute le traitement de chaque *paramètre*.
- **Détecte les alarmes**
  La fonction *alarmesFct()* surveille les mesures corrélées entre elles et émet un message vers le gestionnaire d'alarmes (§9) si certaines conditions se réalisent. Elle détectera, par exemple, l'arrêt du moteur de la machine en surveillant la mesure correspondant à son état. Cette fonction ne traite pas les alarmes sur les valeurs des mesures (dépassement, valeur non initialisée, etc.), ces dernières étant mises à jour dès la "mise en trace", par la *trace* elle même (§10).
- **Exécute les fonctions de sécurité**
  La fonction *securitesFct()* gère les sécurités intermédiaires entre les sécurités matérielles et le comportement graphique des Interfaces Homme-Machine (IHM). En partant de mesures locales ou distantes sur un autre concentrateur, cette fonction exécute une commande. Elle coupera, par exemple, le faisceau à l'entrée de la machine si le moteur s'arrête.
- **Met à jour les status des mesures**
  La fonction *mesureAccesOn()* décide des mesures à remonter vers les écrans graphiques, en fonction d'un certain nombre de critères. Elle contrôle depuis combien de temps la mesure n'a pas été remontée. Si ce temps est supérieur à celui déterminé par le programmeur (100 ms), elle autorise la remontée. Sinon, elle contrôle que la valeur de la mesure ait varié suffisamment depuis sa dernière remontée. Les bits de droit d'accès au *paramètre* (§2.6.4) sont mis auparavant à jour avec les valeurs : PARAM_TO_GMS pour autoriser la remontée du *paramètre*

vers les écrans graphiques, PARAM_INCIDENT et PARAM_ALARME pour signaler respectivement si la valeur du *paramètre* est incidente ou elle a dépassé les seuils d'alarme (inférieur ou supérieur) autorisés.

- **Remonte les mesures**
  La fonction *mesureEcrans()* (§6.2.4) réalise une série d'appels à des fonctions spécifiques à chaque écran (*mesureEcran()*) pour remonter les mesures vers tous les écrans de contrôle ouverts.
- **Re-initialise les mesures**
  La fonction *mesureAccesOff()* place les mesures dans un état tel qu'elles soient toutes considérées comme étant lues. Elle met les bits des droits d'accès au *paramètre* (§2.3.4) PARAM_TO_GMS, PARAM_INCIDENT et PARAM_ALARME à 0.

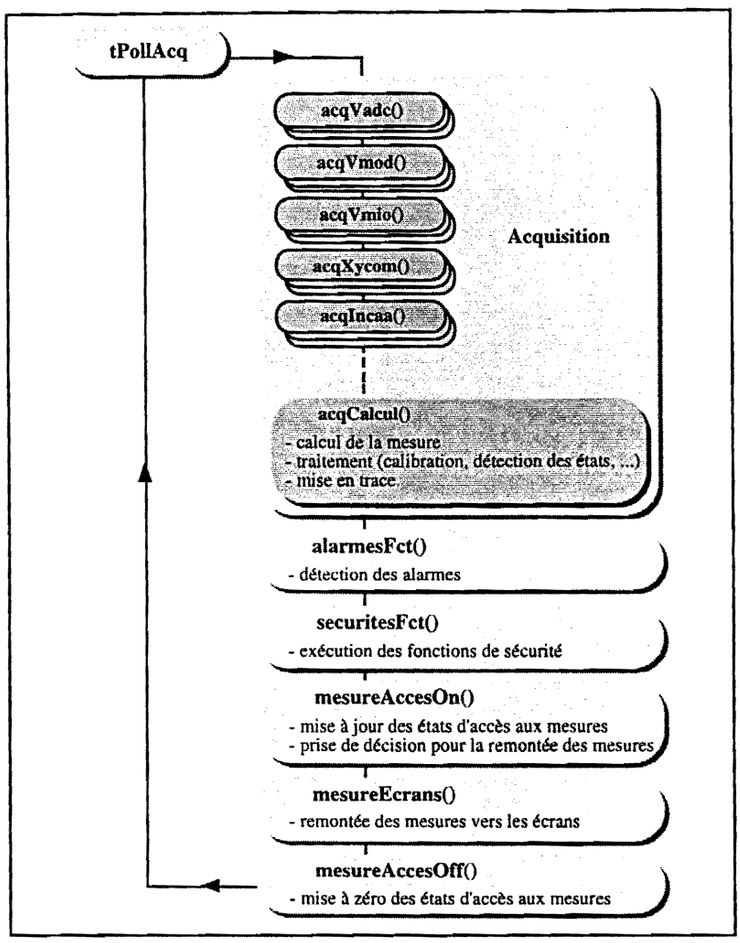


Figure C.10 - La tâche d'acquisition

La boucle d'acquisition ne doit pas être perturbée par une demande d'acquisition extérieure. Cette demande est effectuée soit par l'exécution d'une tâche (tâche de commande, par exemple) soit par une fonction exécutée sous interruption. Elle est prioritaire sur toute acquisition en cours. L'acteur externe doit demander à *tPollAcq* de terminer sa fonction en cours et de revenir immédiatement au début de la séquence exécutée à chaque boucle. Il ne faut pas cependant, inhiber les droits d'accès de la mesure en

cours (*mesureAccesOff()*) alors qu'elle n'a pas encore été remontée (figure C.11).

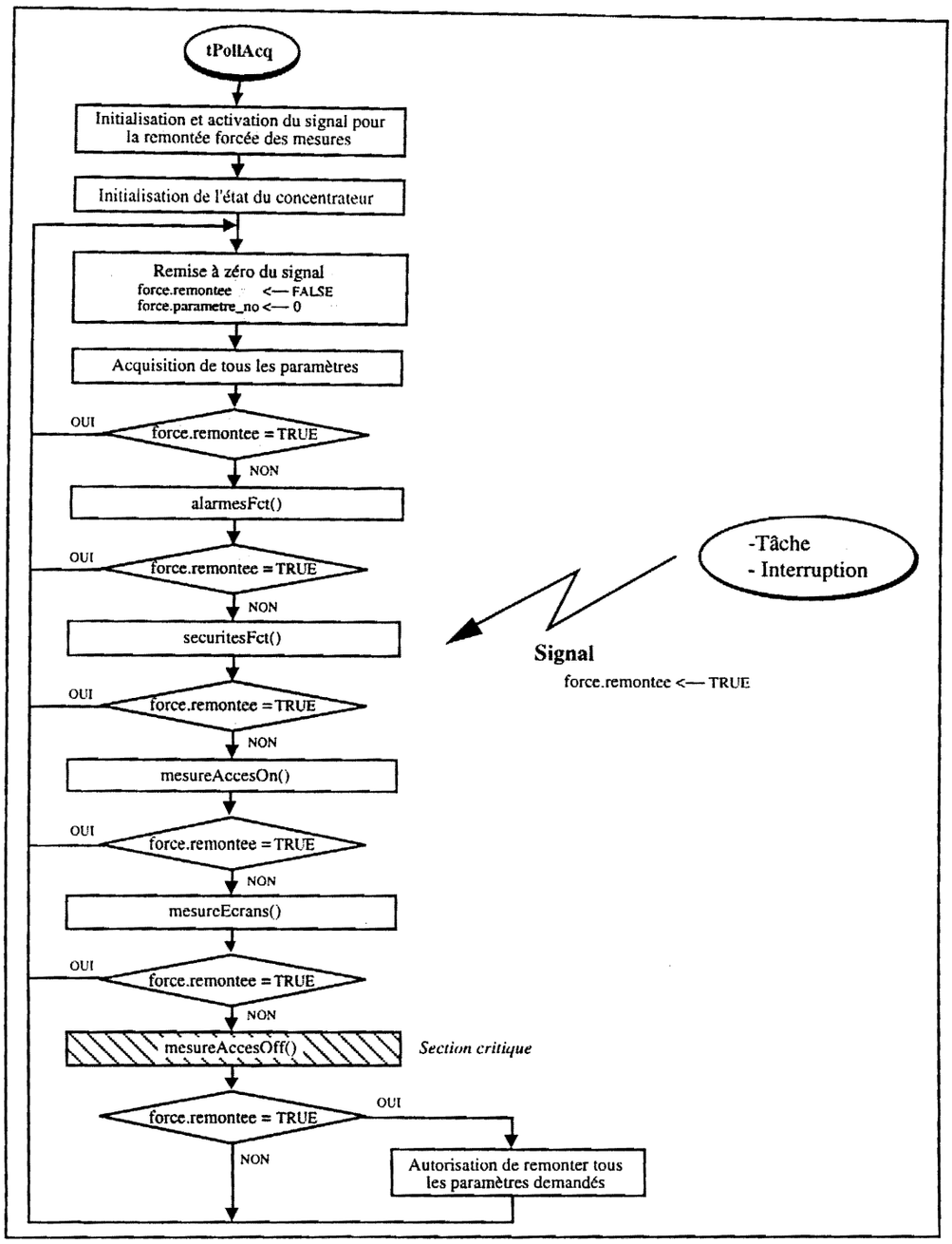


Figure C.11 - Insertion du système de signalisation dans la tâche d'acquisition

Nous ne sommes pas dans le contexte d'un partage de ressource critique car l'acteur externe a une demande prioritaire par rapport à *tPollAcq*. Nous aurions pu, cependant, utiliser les sémaphores d'exclusion mutuelle mais leur mise en place aurait été trop lourde à gérer. Nous avons finalement adopté le mécanisme de signalisation (chapitre B, §3.1.2.2) pour communiquer avec *tPollAcq*. Nous utiliserons le signal numéro 30 (**SIGUSR1**) créé pour l'utilisateur et défini dans le fichier *signal.h* de VxWorks. A ce signal, sera attaché une variable globale de type ***Force*** contenant :

- *un indicateur sur la remontée du (ou des) paramètre(s). Il est actif dès qu'il y a une demande d'acquisition extérieure ;*
- *le nombre de paramètres que l'acteur externe demande à remonter ;*
- *un tableau contenant la structure de données de ces paramètres.*

Chaque processus voulant forcer la remontée d'un *paramètre* devra utiliser la fonction *forceParamToGMS()*. La figure C.11 montre le traitement de ce signal par *tPollAcq*.

3.2 La fonction de calcul : *acqCalcul()*

Une mesure doit être calculée et traitée pour qu'elle puisse être interprétée par l'opérateur au niveau des écrans de contrôle. La valeur prise par le paramètre est affectée lors de l'exécution de la **fonction de calcul** acq*Calcul()*.

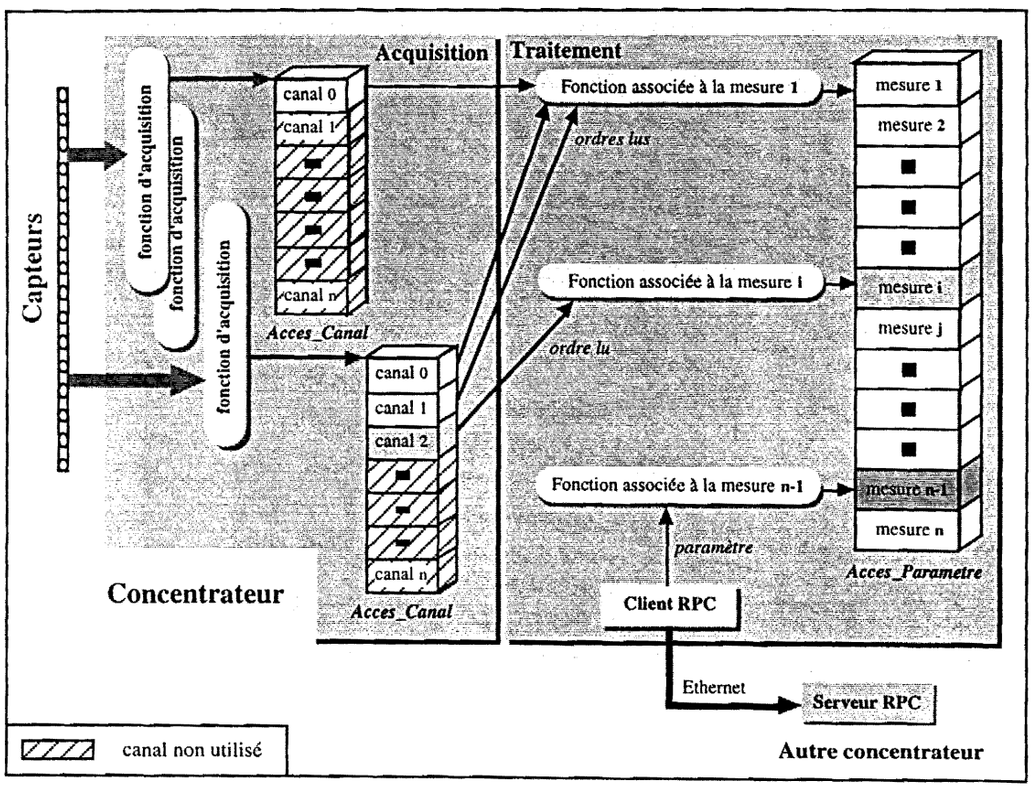


Figure C.12 - Les mesures d'un concentrateur

Tous les *paramètres* associés à un *concentrateur* sont accessibles par une table de pointeurs référencée par une variable globale de type ***Acces_Parametre*** : *accesCalcul* (figure C.12). La *fonction de calcul* parcourra cette table et pour chaque *paramètre* elle :

- exécutera la fonction associée à la mesure (§3.3);
- calibrera et datera la valeur de la mesure ;
- mettra à jour le mot d'état de la valeur de chaque mesure (alarme, débordement, etc.) (§2.6.4);
- fera la "mise en trace" (§10), pour conserver les valeurs pendant une durée déterminée.

3.3 Les fonctions associées aux mesures

Ces fonctions, appelées par la *fonction de calcul*, vont créer les mesures à partir d'*ordres lus* sur un même *concentrateur* et/ou de *paramètres* lus sur n'importe quel *concentrateur* (figure C.12). Il existe plusieurs types de mesures qui sont calculées différemment suivant leur fonctionnalité comme par exemple :

- **Le stripper feuille**
  Le numéro et la position (angle) de la feuille par rapport au faisceau sont obtenus grâce à un encodeur de position absolue qui fournit une information digitale sur 16 fils. Ces 16 états sont lus à l'aide d'une carte d'acquisition de type VMOD (chapitre B, §4.2.2.3). La fonction de mesure donne, après calcul, une valeur comprise entre 0 et 65535.
- **Les mesures TOR**
  Elles sont effectuées par scrutation d'une interface VMOD. Cette interface travaille en logique négative. La valeur lue sera transformée par la fonction de mesure en logique positive.

L'accès aux *paramètres* et aux *ordres* sera contrôlé lors de la génération de code de O_2 par des **alias** (§11.3.1).

3.4 Les fonctions d'acquisition

Les informations fournies par les capteurs sont, soit de nature **analogique** (tension, fréquence, intensité, etc.), soit de nature **Tout Ou Rien** (TOR). Celles fournies par les automates sont une **suite de caractères** lue sur un port série. Ces informations sont traitées par différents types de cartes d'interface, ayant chacune sa propre fonction d'acquisition. Nous conserverons le même principe que pour la première version, c'est-à-dire que nous effectuerons les acquisitions carte d'interface par carte d'interface plutôt que mesure par mesure (§1.1).

```
acqxxx()

Début
    Acquisition <-- ON;
    Si (accès carte = OFF ) OU (nombre de canaux = 0) alors
        Acquisition <-- OFF ;
        Fin de la fonction d'acquisition;
    FinSi
    Pour chaque canal faire
        Si accès au canal = FALSE alors
            Faire l'acquisition matérielle;
            Stocker dans la table d'accès aux canaux
            la valeur et la date de l'acquisition;
            Sélectionner le canal suivant;
        FinSi
        Acquisition <-- OFF;
    FinPour
Fin
```

Figure C.13 - Principe d'une procédure d'acquisition

A chaque carte d'interface nous allons associer une table pour accéder aux valeurs de tous les canaux

de la carte (utilisés ou pas). Cette table d'accès aux canaux sera référencée et initialisée par une variable globale de type ***Acces_Canal*** et sera allouée en mémoire au démarrage du concentrateur. Chaque élément de la table contiendra une valeur datée et codée en flottant double, et l'état du canal (utilisé ou pas utilisé). La fonction d'acquisition ne remplira dans cette table que les éléments associés aux canaux utilisés (figure C.12). Cette méthode permet de ne pas faire d'acquisition sur les canaux non utilisés.

Les fonctions d'acquisition **ne sont pas chargées de prendre des décisions** suite à une acquisition (alarme, interdiction de commande, etc.), cela étant réservé à d'autres modules. **Leur traitement doit rester au plus près du matériel**. Elles vont lire séquentiellement tous les canaux de la carte, et convertir chaque information lue sur un capteur ou un automate en une donnée numérique datée et codée sur un flottant double. Un drapeau indique le début et la fin de l'acquisition (figure C.13). Il se peut qu'une tâche extérieure vienne lire un canal en cours d'acquisition. Pour éviter toute confusion créée par un accès concurrent à un même canal, cette tâche utilisera le mécanisme de signalisation (§3.1)

4 Traitement des commandes

Les commandes sont matérialisées au niveau de GMS soit par un curseur pour une commande de type analogique soit par un bouton pour une commande de type TOR. Elles doivent permettre :

- d'activer ou de désactiver un actionneur de type TOR ;
- d'assurer des fonctionnalités propres aux commandes analogiques (la montée en courant d'une alimentation d'aimant, par exemple).

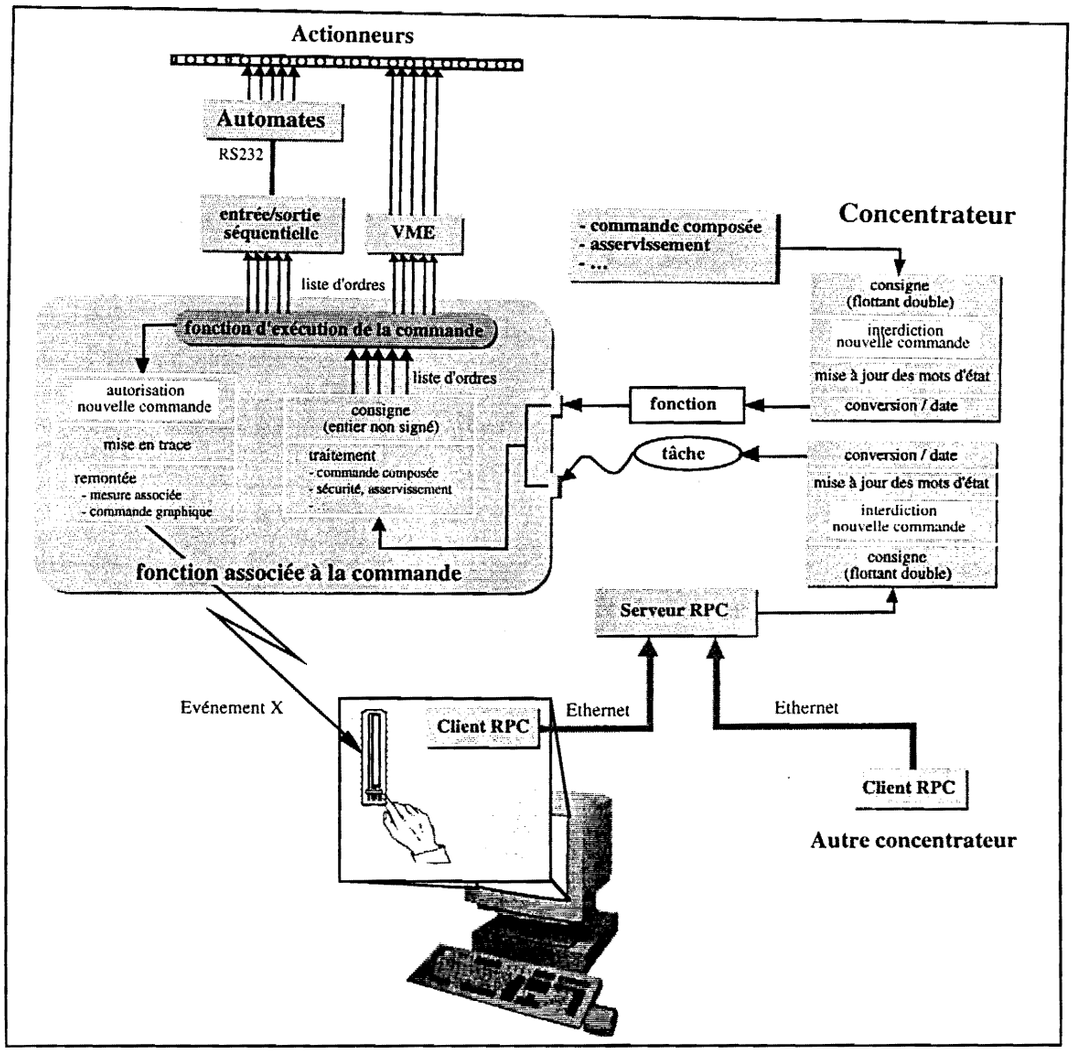


Figure C.14 - Le traitement des commandes

Une commande est déclenchée soit par le serveur RPC de commande, soit par un mécanisme interne tel qu'un asservissement ou une sécurité (figure C.14). L'exécution d'une commande se résume à l'exécution de la fonction associée au paramètre. Elle est effectuée sous forme de tâche indépendante quand la commande est demandée par le serveur RPC de manière à ne pas bloquer ce dernier. Elle peut être, aussi, effectuée sous la simple forme d'un appel à une fonction pour un asservissement ou une commande composée. Le mot d'état de la valeur du *paramètre* commandé est positionné

(VALEUR_NOT_READY), interdisant toute nouvelle commande jusqu'à ce que la **fonction d'exécution de la commande** soit achevée. Cela, afin d'assurer intégralement et sans discontinuité l'exécution de la commande.

4.1 Les fonctions associées aux commandes

Elles vont interpréter les commandes sous forme d'une suite d'*ordres lus* ou une suite d'*ordres écrits* en provenance, ou en direction, d'un même *concentrateur* ou de *concentrateurs* différents. Ce sont ces fonctions qui vont prendre des décisions suivant la nature de la commande :

- **Les impulsions à durée variable**
  Elles sont, en général associées à un bouton **On/Off**. Elles permettent l'arrêt (impulsion sur le **Off**) ou la mise en marche (impulsion sur le **On**) d'un moteur, le *reset* d'un automate (impulsion sur le **On**, puis **Off**), etc.
- **L'arrêt d'une alimentation**
  Avant d'éteindre une alimentation en courant, il faut ramener son intensité à 0. L'opérateur exécute une commande d'arrêt non bloquante. Le serveur RPC lance la fonction associée à cette commande sous forme de tâche et se libère. La fonction associée, à son tour, fait appel à une commande bloquante qui fera chuter le courant. Un fois cette commande réalisée, elle fait un **Off** sur l'alimentation.
- **Une sécurité**
  Ce type de commande peut interdire ou autoriser une commande sur le même ou sur un autre concentrateur.
- **Le stripper feuille ou les mesures TOR**
  Voir §3.3.

5 Les cartes VME et les automates

5.1 Définition d'un *lien*

On appelle **lien** toute interface entre le *concentrateur* et les équipements. Ainsi, un **lien VME** est une carte VME (VDAC, VADC...) et un **lien série** est un automate (INCAA, BALZERS...). A chaque *lien* est associé un symbole représentant l'adresse VME pour les cartes VME et le port série pour les automates. Ce symbole est constitué du type de *lien* suivi de son numéro (exemple : le premier *lien* série INCAA sera nommé **Incaa_0**). Les caractéristiques d'un *lien* sont modélisées par une structure de données de type ***Adresse_Lien*** comportant :

- *le symbole du lien ;*
- *l'accès au lien (LIEN_ACCES_ON ou LIEN_ACCES_OFF) ; l'accès est interdit si le lien est absent ou défectueux ;*
- *l'état de l'acquisition (ON ou OFF) ; c'est le drapeau qui indique le début et la fin de l'acquisition ;*
- *le descripteur de fichier correspondant au port série ouvert ;*
- *le nombre de canaux fictifs pour le lien série (§5.4) ;*
- *un pointeur sur le tableau des canaux fictifs (§5.4) ;*
- *le nom du port série ou l'adresse VME.*

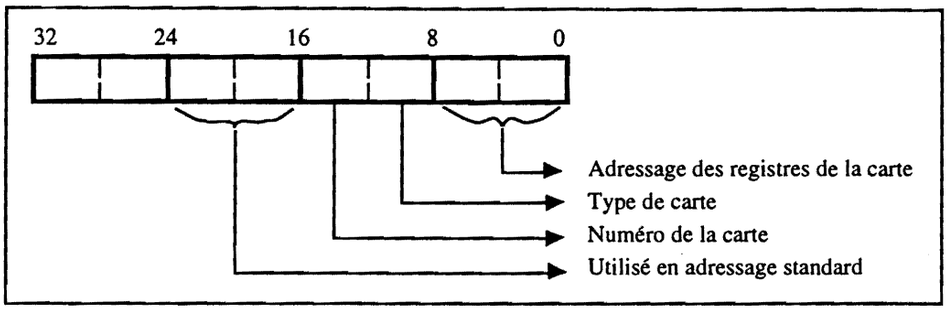


Figure C.15 - Norme d'adressage

Interface	Adressage	Code modificateur d'adresse	Adresse de base
VDAC	court	VME_AM_USR_SHORT_IO	0x1000
VADC	court	VME_AM_USR_SHORT_IO	0x1200
VMOD	standard	VME_AM_STD_USR_DATA	0xFE1400
Moteur	standard	VME_AM_STD_USR_DATA	0x801600
XYCOM	court	VME_AM_USR_SHORT_IO	0x1800
VMIO	court	VME_AM_USR_SHORT_IO	0x1C00
GPIB	court	VME_AM_USR_SHORT_IO	0x1E00

Tableau T.7 - L'adressage des cartes VME

En ce qui concerne les *liens VME*, ils utilisent un adressage court, le plus souvent et un adressage long (chapitre B, §2.3.1) dans certains cas particuliers. La figure C.15 nous montre la définition retenue pour les adresses VME des cartes d'interface et le tableau T.7 nous donne le type d'adressage et l'adresse de base par type de *lien VME*. Dans un même type de carte, l'adresse de base est incrémentée de 0x1000 pour toute carte rajoutée.

Automate	Nom du port	Notation utilisateur
réservé au système	"/tyCo/0"	console
BALZERS ou TPG300	"/tyCo/1"	port série n° 2
Teslamètre	"/tyCo/2"	port série n° 3
INCAA ou INCAA18	"/tyCo/3"	port série n° 4

Tableau T.8 - L'attribution des numéros de port pour les automates

En ce qui concerne les *liens série*, ils sont reliés à l'un des ports série de la carte CPU placée dans le châssis VME. Cette carte comporte 4 ports, le port "**/tyCo/0**" étant réservé au système. Chaque automate peut être différencié lors de sa détection sauf INCAA et INCAA18 qui ont un protocole de communication identique. Mais ces deux derniers *liens* ne cohabitent jamais dans le même châssis VME. Tout *lien série* aura donc la possibilité de choisir parmi les trois ports disponibles ("/tyCo/1", "/tyCo/2" et "/tyCo/3"). Toutefois, nous avons décidé d'attribuer un numéro de port à chaque type d'automate (tableau T.8).

5.2 Auto-configuration des *liens*

Nous avons créé un module d'auto-configuration par type de carte et d'automate pour détecter tous les *liens* présents dans les *concentrateurs* lors du démarrage du système. Si un *lien* est absent l'auto-configuration ne sera pas interrompue et son symbole sera créé. L'accès à ce *lien* sera simplement interdit (LIEN_ACCES_OFF). Ce principe permet de travailler en mode dégradé et de ne pas être handicapé si un *lien* est défectueux. Les figures C.16a et C.16b montrent le principe des modules d'auto-configuration pour les *liens VME* et les *liens série* respectivement.

```
autoConfTypeLienVME()

Début
    Si nombre de cartes = 0 alors
        Fin de la fonction d'auto-configuration;
    FinSi
    Initialisation de la structure Adresse_Lien (adresseLienInit());
    Calcul de l'adresse VME (sysBusToLocalAdrs());
    Pour toutes les cartes présentes de même type faire
        Détection de la carte (detectxxx());
        Si la carte est détectée
            Alors accès au lien VME <-- ON;
            Sinon accès au lien VME <-- OFF;
        FinSi
        Création du symbole de la carte;
    FinPour
Fin
```

Figure C.16a - Principe d'auto-configuration pour un lien VME

```
autoConfTypeLienSerie()

Début
    Si nombre d'automates = 0 alors
        Fin de la fonction d'auto-configuration;
    FinSi
    Pour tous les automates présents de même type faire
        Initialisation de la structure Adresse_Lien (adresseLienInit());
        port série <-- "/tyCo/n";  /* 1 < n < nombre de cartes+1 */;
        Si le port est déjà attribué
            Alors aller au début de la boucle Pour;
            Sinon détection de l'automate (detectxxx());
                Si l'automate est détecté
                    Alors Attribution du port;
                        Accès au port <-- ON;
                        Création du symbole de l'automate;
                    Sinon retourner message d'erreur;
                FinSi
        FinSi
    FinPour
    Pour tous les automates qui n'ont pas été détectés faire
        Accès au port <-- OFF;
        Création du symbole de l'automate;
    FinPour
Fin
```

Figure C.16b - Principe d'auto-configuration pour un lien série

5.3 Détection des liens

```
detectTypeLienSerie()

Début
    Ouverture du port série (open()) et stockage du descripteur de fichier dans la structure Adresse_Lien;
    Initialisation du timeout en lecture à 1 seconde;
    Initialisation du port série (ioctl());
    Attente de 20 secondes pour laisser le temps au "chip IO" de la carte CPU de se configurer;
    Le tampon de lecture du port série est vidé (ioctl(), read());
    Répéter 3 fois
        Emission du message de reconnaissance (read());
        Attente de 250 ms;
        Répéter 3 fois
            Lecture de la réponse (read());
            Si le timeout de la lecture (select()) a expiré
                Alors refaire la lecture;
                Sinon décoder le message caractère par caractère
                    Si le message décodé est reconnu alors
                        Le lien série est détecté;
                        Fin de la fonction de détection;
                    FinSi
            FinSi
        FinRépéter
    FinRépéter
    Le port série n'est pas détecté;
    Fermeture du port série;
Fin
```

Figure C.17 - Principe du module de détection des liens série

Au démarrage, le *concentrateur* doit détecter tous les *liens* qui lui ont été attribués, chaque type de *lien* ayant un module de détection appelé *detectxxx()*. La détection d'un *lien VME* est relativement simple. Il suffit de lire (carte programmée en entrée) ou d'écrire (carte programmée en sortie) sur un registre bien défini en utilisant la fonction VxWorks d'Entrée/Sortie : *VxMemProbe()*. La détection d'un *lien série* est beaucoup plus complexe.

L'automate est connecté sur l'un des ports série de la carte CPU placée dans le châssis VME. Pour communiquer avec lui, le port doit être ouvert en Lecture/Ecriture et configuré avec les bons paramètres de communication (vitesse de transmission, parité...). Cela étant fait, nous lui envoyons un message type et nous attendons la réponse. Si nous communiquons avec le bon automate, celui-ci répond avec un message connu et nous laissons le port ouvert et configuré. Sinon, le port est fermé pour qu'il puisse être utilisé dans la détection d'un autre automate. Ce principe est décrit par l'algorithme de la figure C.17.

5.4 Communication avec les *liens*

La communication avec un *lien VME* se résume à l'accès à une adresse VME. En effet, l'adresse VME de la carte est utilisée comme un pointeur sur une structure de données spécifique à cette carte qui permet d'accéder, simplement, aux canaux physiques du *lien*. Par contre, la communication avec un *lien série* se résume à l'échange de messages sous forme d'une suite de caractères, sur un port série. Ces messages nous renseignent sur les différents états de l'automate (mise sous tension, type d'erreur, mesure d'un paramètre, etc.).

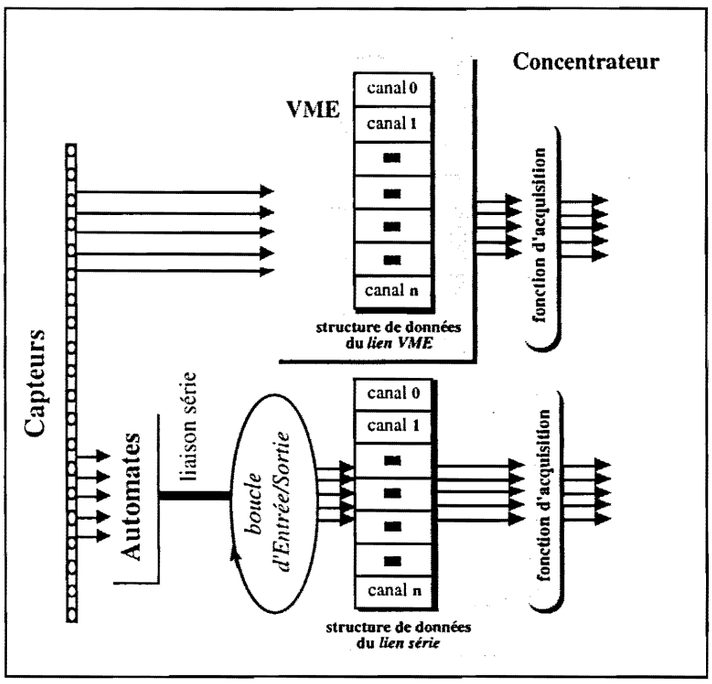


Figure C.18 - La communication avec les liens série et VME

Pour rester homogène avec les *liens VME,* par convention, **chaque état du lien série représentera un canal fictif.** Tous les canaux sont regroupés dans une structure de données spécifique au type de *lien série*. Cette structure, allouée dynamiquement en mémoire lors de l'initialisation du *lien*, est mise à jour régulièrement par une **boucle d'Entrée/Sortie** (figure C.18). La boucle d'Entrée/Sortie, exécutée par une tâche, n'utilise plus les gestionnaires standard (§1.1) mais des fonctions travaillant en interruption (*read(), write()* ...) (figure C.19). Cette méthode permet une communication plus souple avec l'automate (gestion des *timeouts*, lecture non bloquante...) et un remplissage plus rapide de la structure de données.

La boucle d'Entrée/Sortie effectue cycliquement les fonctions suivantes :

- émission du premier message et réception de la réponse ;
- émission du message suivant, réception de la réponse et ainsi de suite jusqu'au dernier message ;
- décodage de tous les messages et remplissage de la structure de données.

Du fait que le *lien série* est constamment en dialogue avec la tâche d'Entrée/Sortie, il est difficile pour une tâche extérieure (en l'occurrence une tâche de commande) de communiquer avec ce *lien* sans rentrer en conflit avec la tâche d'Entrée/Sortie. Pour éviter cette situation, le module d'émission-réception des messages est protégé par un sémaphore d'exclusion mutuelle non bloquante. Ainsi, tant que l'automate n'a pas répondu au message qui lui a été envoyé, aucun autre message, quelque soit son origine, ne peut lui être adressé. La figure C.19 nous montre le principe du module d'émission/réception d'un message.

```
sendRecvMsgTypeLienSerie()

Début
    Récupération du descripteur de fichier du port série ouvert par le module detectTypeLienSerie();
    Initialisation du timeout en lecture à 1 seconde;
    Le tampon de lecture du port série est vidé (ioctl(), read());
    Activation du sémaphore d'exclusion mutuelle avec un "timeout" de 2 secondes (semTake());
    Répéter 3 fois
        Emission du message (write());
        Attente de 250 ms;
        Répéter 3 fois
            Lecture de la réponse (read());
            Si le timeout de la lecture (select()) a expiré alors
                Refaire la lecture;
            Sinon décoder le message caractère par caractère;
                Si le message décodé n'est pas reconnu alors
                    Erreur d'Entrée/Sortie;
                FinSi
                Désactivation du sémaphore d'exclusion mutuelle (semGive());
                Fin de la fonction;
            FinSi
        FinRépéter
    FinRépéter
    Le port série ne répond plus;
    Désactivation du sémaphore d'exclusion mutuelle (semGive());
Fin
```

Figure C.19- Le module d'émission/réception d'un message

6 La communication écrans GMS - *concentrateurs*

Le premier système a été très influencé par la particularité de GMS (chapitre B, §3.4.3). Les mesures étaient traitées par scrutation périodique et les *concentrateurs* manquaient d'autonomie. Ils ne pouvaient pas signaler leur état de fonctionnement eux-mêmes, ce qui interdisait à l'opérateur d'avoir une image de l'état du système à un instant donné.

L'idée est de ne plus utiliser les RPC comme support de communication entre les écrans et le *concentrateurs*, mais le support adopté par les applications GMS : les événements X. Pour déclencher le traitement de son application, GMS utilise un événement X de type *gmstimer* (chapitre B, §3.4.3). Pour traiter la communication avec les *concentrateurs*, nous allons agir de même, en créant nos propres types d'événements X (figure C.20).

La communication par événements X va nous permettre d'atteindre trois objectifs importants :

- autoriser les reprises sur panne et répercuter les commandes sur tous les écrans "jumeaux" par le biais des **formulaires** (§6.3) ;
- refléter l'état réel de fonctionnement du système par le biais des **balises** (§6.4) ;
- améliorer la dynamique des écrans par le biais des événements de type **EV_MESURE**.

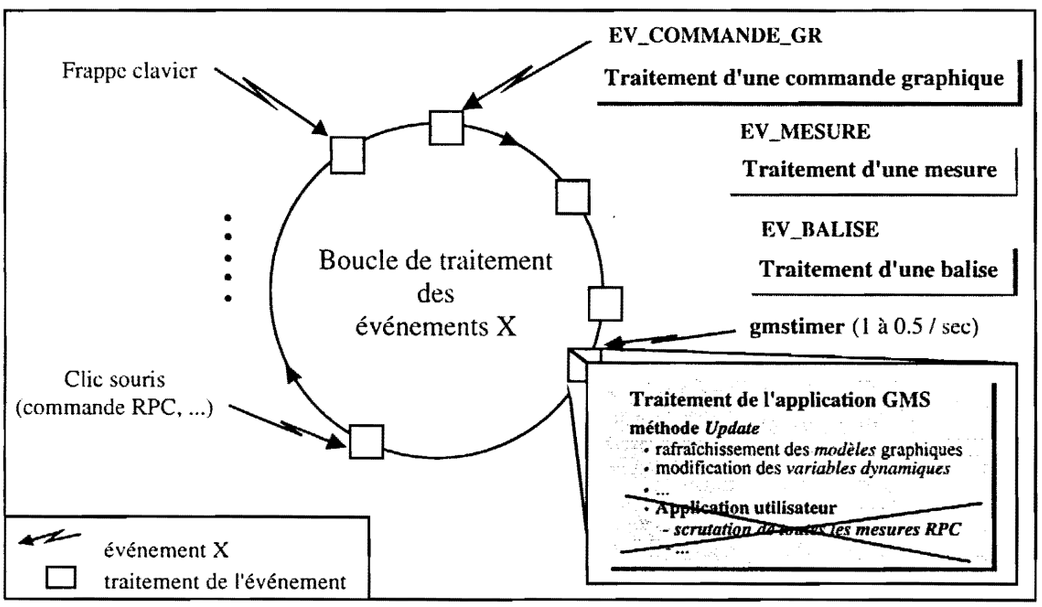


Figure C.20- La nouvelle boucle GMS

L'événement de type **EV_MESURE** traite la remontée des mesures vers les écrans de contrôle. La prise en compte des mesures est, ainsi, insérée dans la boucle de traitement des événements X de GMS au même titre qu'une commande ou qu'un changement de taille d'une fenêtre. La scrutation périodique est supprimée et GMS peut travailler en serveur, les *concentrateurs* étant ses clients.

Lorsque l'opérateur passe une commande au niveau des écrans de contrôle il faut qu'elle soit répercutée sur tous ses écrans jumeaux. Pour cela, nous avons créé l'événement de type

EV_COMMANDE_GR qui contient la consigne de la commande. Cet événement est envoyé, par la fonction associée à la commande (§4.1), à tous les écrans concernés (§6.2.2). Cette fonctionnalité n'existait pas dans la première version car son adaptation aurait été trop complexe. Ce manque était surtout ressenti au démarrage d'un écran de contrôle. Les commandes n'étaient pas mises à jour et, donc, l'écran ne reflétait pas l'état réel des commandes.

Le protocole X nous permet de rendre les *concentrateurs* autonomes et d'alléger l'architecture de notre système, même s'il n'a pas été créé pour le transfert de données. Le système n'est plus interrogé depuis le pupitre de la machine mais ce sont les *concentrateurs*, eux-mêmes, qui signalent leur état et remontent les mesures vers les écrans graphiques. Pour cela, des clients X sont lancés sur chaque concentrateur, chacun étant le correspondant d'un écran GMS. Ceux-ci envoient, par messages intégrés dans le flux des événements X, toutes les informations nécessaires à l'affichage d'un *paramètre*. Ce mécanisme, qui va être développé dans les paragraphes suivants, est représenté sur la figure C.21.

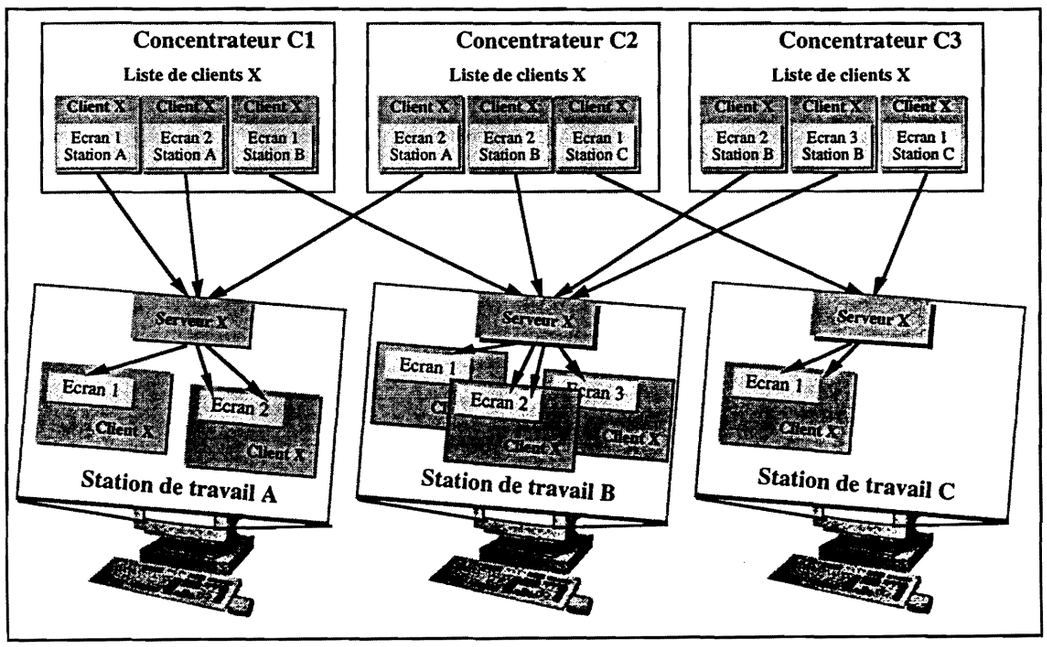


Figure C.21 - Communication écrans - concentrateurs

6.1 Les écrans GMS

6.1.1 Ergonomie retenue

Nous avons choisi pour l'Interface Homme-Machine (IHM) d'utiliser exclusivement des visualisations et des actions basées sur l'utilisation d'un interface graphique (souris, écrans). La saisie d'une consigne par le clavier exigerait, pour garantir la cohérence des commandes envoyées ou pour éviter les erreurs de frappe, un dispositif de confirmation des commandes alourdissant terriblement

l'ergonomie. De ce fait, nous avons totalement exclu cette saisie de notre système. Les consignes se font soit par des curseurs soit par des boutons (marche-arrêt ou flash). Nous avons reconfiguré les boutons de la souris pour avoir la possibilité de faire des réglages fins (figure C.22). Dans la même idée, nous avons développé un curseur doté de plusieurs fonctionnalités. Les réglages classiques sont faits en déplaçant le curseur à l'aide de la souris. Si l'utilisateur veut un réglage référencé, il lui suffit de déplacer un repère (figure C.22) jusqu'à la valeur souhaitée. En cliquant dans la zone réservée pour ce repère, le curseur se déplace automatiquement jusqu'à la valeur de celui-ci. Enfin, en cliquant au dessus (ou au dessous) du curseur on le déplace de un pas vers le haut (ou vers le bas). Si l'utilisateur a cliqué avec le bouton de réglage fin, le pas sera 1/10 du pas standard (valeur réglable dans la base de données). Quant aux mesures, elles sont visualisées soit par des thermomètres gradués soit par des afficheurs numériques.

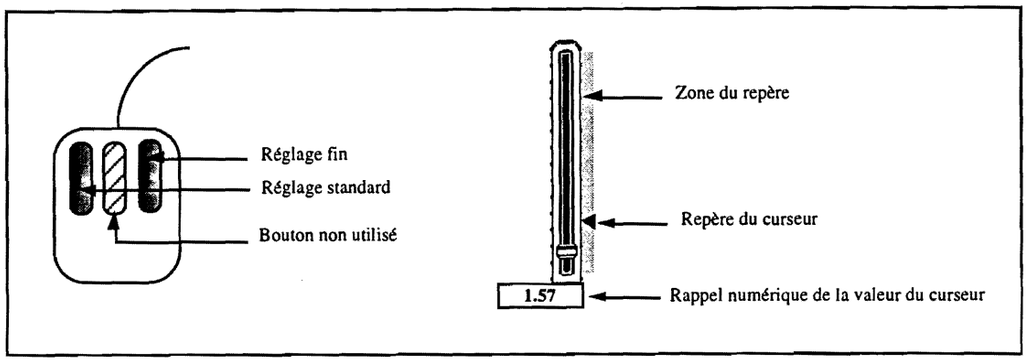


Figure C.22 - La souris et le curseur dans une application GMS

Une telle limitation n'influence en rien la richesse de l'interface. Elle permet, plutôt une grande homogénéité et une généralisation immédiate à tout le système de toute amélioration apportée.

Pour commander et contrôler totalement l'accélérateur Vivitron nous avons décomposé la machine en trois grandes parties, contenant chacune plusieurs éléments (tableau T.8). Chaque élément est visualisé sur un écran GMS et représente une partie fonctionnelle du Vivitron.

Partie de l'accélérateur	Nom des écrans
Injecteur	Source 860 Extension BE Pulsation
Générateur	Générateur Système de Charge Intérieur Machine
Ligne du faisceau	Tronc Commun Lignes Spécifiques Vide (Groupe de Pompage) *Faraday Cups* - Vannes Vide - Lèvres

Tableau T.8 - Les différents écrans du système de contrôle et commande

L'écran *Source 860* permet de régler le faisceau d'ions au niveau de la source disposée sur l'une des

trois voies (chapitre A, §1.3.2.1). Le conditionnement et le transport du faisceau jusqu'à l'entrée de la machine est effectué par l'écran *Extension BE*. Pour chaque voie il y a une paire d'écrans et la paire ouverte sera celle qui correspondent à la voie utilisée. L'écran *Pulsation* assure les fonctionnalités de la pulsation des ions en BE.

L'écran *Générateur* permet de démarrer les moteurs qui entraînent la courroie et les alternateurs. Il permet la montée en tension du Vivitron en réglant les alimentations HE et BE. Il mesure les courants dans les résistances tubes et colonnes de SM0 et SM18 et dévalide ou revalide la fonction de stabilisation du faisceau. L'écran *Système de Charge* contrôle les bilans de charge de la machine et l'écran *Intérieur Machine* ajuste la trajectoire du faisceau pour assurer une focalisation optimale du faisceau à la sortie de la machine.

Les écrans *Tronc Commun* et *Lignes Spécifiques* dirigent le faisceau depuis la sortie de la cuve jusqu'aux différentes cibles. L'écran *Vide* et ses sous-écrans (*Groupe de Pompage*) donnent l'état du vide sur l'ensemble du site et commandent les différentes pompes. L'écran *Faraday Cups* permet de couper le faisceau à différents endroits de la ligne de faisceau.

6.1.2 Représentation d'un concentrateur dans une application GMS

Un *concentrateur* est décrit sur les écrans de contrôle par une structure de données appelée ***Conc_Struct*** et contenant :

- *le nom du concentrateur ;*
- *son état : information transmise par les balises (§6.4) ;*
- *un flag indiquant si le concentrateur est utilisé par l'écran ou non ;*
- *les paramètres pour la vérification de la communication écran - concentrateur : nombre de fois que la connexion est vérifiée (verif_cmpt), la date de la vérification en cours et de la vérification précédente, nombre de boucles de tPollAcq lors de la vérification en cours et de la précédente vérification (§3.3.1) ;*
- *la date d'arrivée de la dernière balise (§6.4) ;*
- *le nombre d'essais de connexion ;*
- *une liste de pointeurs sur les formulaires (§6.3) des mesures GMS et le nombre de mesures ;*
- *une liste de pointeurs sur les formulaires (§6.3) des commandes GMS et le nombre des commandes.*

6.1.3 Définition d'un écran GMS

Un écran GMS est représenté sur les écrans de contrôle par une fenêtre graphique (chapitre B, §1.3.3.2). Il est modélisé par une structure de données appelée ***Ecran_Struct*** et contenant :

- *le nom de l'écran ;*
- *le mode d'ouverture (commande, mesure ou test) (§8) ;*
- *le nombre de mesures GMS composant l'écran et un tableau contenant ces mesures ;*
- *le nombre de commandes GMS composant l'écran et un tableau contenant ces commandes ;*
- *le display associé à cet écran ;*

- *la liste des formulaires (§6.3) des mesures GMS et le nombre de mesures ;*
- *la liste des formulaires (§6.3) des commandes GMS et le nombre de commandes ;*
- *le nombre de concentrateurs.*

6.1.4 Définition d'une application GMS

Une application GMS est le programme d'application utilisateur qui gère tous les écrans et sous-écrans GMS. Elle est constituée par une fenêtre mère (chapitre B, §1.3.3.2) qui gère toutes les fenêtres des écrans GMS. Elle est modélisée par une structure de données appelée ***Appli_Struct*** et contenant :

- *les références du client X associé à la fenêtre mère de l'application : le nom et la structure de données du display, et l'identificateur de fenêtre ;*
- *les références du gestionnaire d'écrans xvivetat (§8) : le display et l'identificateur de fenêtre ;*
- *le nombre de concentrateurs associés à l'application*
- *le nombre total d'écrans associés à l'application et le nombre d'écrans ouverts ;*
- *une liste de pointeurs sur les écrans GMS ;*
- *la liste des concentrateurs.*

Tous les traitements concernant l'application GMS (rafraîchissement des modèles graphiques, modification des variables dynamiques, traitement des événements X, etc.) vont s'effectuer à ce niveau pour être répercutés ensuite au niveau des écrans GMS. Une application GMS se déroule principalement en trois phases :

- Initialisation de GMS :
  - *gmsSetup()* : initialise GMS et appelle les fonctions d'initialisation utilisateur ;
  - *gmsMainInit()* : crée le gestionnaire de fenêtres GMS et initialise les objets graphiques ;
  - *gmsInitStates()* ;
- Initialisation de l'application :
  - initialisation des variables dynamiques ;
  - installation des classes des écrans et création d'instances ;
  - initialisation utilisateur (*nomApplication_init_fct()*).
- Activation de la boucle principale :
  - *gmsMainLoop()* : gère les événements et la mise à jour des variables dynamiques (§1.1.1).

Dans notre exposé, nous allons passer en revue surtout les modules qui sont affectés par la communication écrans - *concentrateurs*. Comprendre leur fonctionnement nous sera utile notamment lors de l'introduction aux *balises* (§6.4) et aux *formulaires* (§6.3).

6.1.4.1 Initialisation de l'application par l'utilisateur

L'algorithme ci-dessous nous donne un bref aperçu des différentes étapes d'initialisation d'une application GMS par l'utilisateur. Nous remarquerons notamment l'affectation des méthodes **activate**, **update** et **deactivate** à chaque objet écran créé et l'appel à la fonction *interceptXEvent()*. Nous noterons, aussi, la création d'une variable globale, **conc_en_defaut**, contenant la liste des *concentrateurs* "défectueux".

*nomApplication_**init_fct**()*

Début

Initialisation de la structure de données de l'application (appliStructInit());
Association de la fonction utilisateur de gestion des événements X, interceptXEvent() (§6.1.4.5), au gestionnaire de fenêtres GMS (gmsWsLowEventFctn());
Initialisation de la liste des concentrateurs de l'application GMS;
Allocation dynamique en mémoire de la variable globale, conc_en_défaut, contenant la liste des concentrateurs défectueux;
Instanciation des classes écrans et affectation à chaque objet des méthodes "activate", "update" et "deactivate";
Création dynamique de la liste qui va contenir les concentrateurs en défaut dans l'application;
Initialisation de tous les écrans attachés à l'application;
Affectation aux formulaires (§6.3) des données concernant le clients X de chaque écran GMS (appliStructSetFormulairesClientX());

Fin

Figure C.23a - Algorithme d'initialisation utilisateur de l'application

6.1.4.2 Ouverture d'un écran GMS

L'ouverture d'un écran GMS déclenche la méthode *activate* (figure C.23b). Cette méthode effectue, entre autre, le conditionnement de l'écran GMS pour qu'il soit pris en compte par le gestionnaire d'écrans *xvivetat* (§8), ainsi que l'échange des *formulaires* (§6.3) de début de communication avec les *concentrateurs*.

6.1.4.3 Mise à jour d'un écran GMS

Chaque écran GMS est doté d'une méthode "update" (figure C.23c), exécutée périodiquement, qui vérifie surtout la communication écran - *concentrateurs*. Si un *concentrateur* est en défaut une tentative de connexion est faite et s'il répond il y a échange de *formulaires* (§6.3). Si au bout d'un certain nombre de tentatives, le *concentrateur* ne répond toujours pas nous passons au *concentrateur* suivant.

6.1.4.4 Fermeture d'un écran GMS

La fermeture d'un écran GMS déclenche la méthode *deactivate* (figure C.23b). Cette fermeture est signalée au gestionnaire d'écrans *xvivetat* (§8) et un échange de *formulaires* de fermeture (§6.3) est effectué entre l'écran et tous les *concentrateurs* associés.

6.1.4.5 Traitement des événements X par l'utilisateur

Nous avons vu au début de ce chapitre pourquoi nous avons choisi les événements X comme moyen de communication entre les écrans et les *concentrateurs*. Nous avons créer trois types d'événements utilisateur :

- **EV_MESURE** contient des renseignements sur les mesures (§6.2.4) ;
- **EV_COMMANDE_GR** contient des renseignements pour la remise à jour graphique des commandes (§6.2.4) ;
- **EV_BALISE** contient des renseignements sur les *balises* (§6.4).

Pour communiquer entre clients X nous utilisons l'événement de type ***ClientMessage***, avec un message composé de 20 caractères (chapitre B, §1.3.3.4), le premier caractère étant réservé au type d'événement utilisateur. Sous GMS c'est la fonction *interceptXEvent()* (§6.1.4.1) qui intercepte tous

les événements utilisateur et traite leurs données. Pour cela elle vérifie que l'événement reçu par le client X associé à l'application GMS soit bien de type ***ClientMessage*** et que le message contenu dans l'événement soit au format 8 bits.

```
nomEcran_activate()

Début
    Récupération de la structure du display de l'écran GMS (gmsQDisplayId());
    Initialisation du client X associé à cet écran, pour xvivetat (§8);
    Ouverture de tous les écrans GMS de l'application au niveau de xvivetat et mise en service de tous les
    concentrateurs concernés (appliOuvreEcran());
    Insertion de l'écran dans la liste des écrans GMS de l'application (appliStructAddEcran());
    Préparation des formulaires et échange de ceux-ci avec les concentrateurs associés à l'écran
    (ecranStructFormulairesOn()) (§6.3);
Fin
```

Figure C.23b - Algorithme de la méthode activate d'un écran

```
nomEcran_update()

Début
    Détection de tous les concentrateurs qui ont un défaut de communication avec l'application GMS
    (appliStructVerifCommunication() (§6.4.2));
    connect_compt <— 0;   /* nombre de tentatives de connexion */
    Pour chaque concentrateur défectueux faire
        Si connect_compt = 0 alors
            Envoyer le message d'alarme "défaut concentrateur";
        FinSi
        connect_compt = connect_compt + 1;
        Si connect_compt ≥ 15 alors
            connect_compt <— 0;
            Essayer à nouveau la connexion (ping());
            Si le concentrateur ne répond toujours pas
                Alors passer au concentrateur suivant;
                Sinon Envoyer le message d'alarme "concentrateur répond";
                      Echanger les formulaires avec le concentrateur (concFormulairesEchange()) (§6.3);
            FinSi
            Remplir la structure des mesures et des commandes GMS avec les informations obtenues par les
            formulaires;
            Re-inititialiser la structure du concentrateur;
        FinSi
    FinPour
Fin
```

Figure C.23c - Algorithme de la méthode update d'un écran

```
nomEcran_deactivate()

Début
    Préparation des formulaires et échange de ceux-ci avec les concentrateurs associés à l'écran
    (ecranStructFormulairesOff()) (§6.3);
    Fermeture de tous les écrans de l'application au niveau de xvivetat et mise hors service des concentrateurs
    associés à l'application (appliFermeEcran());
    Suppression de l'écran de la liste des écrans GMS (appliStructRetireEcran());
Fin
```

Figure C.23d - Algorithme de la méthode deactivate d'un écran

6.2 Les concentrateurs

6.2.1 L'identificateur du *paramètre*

Dans un événement X, l'espace réservé au transport de données n'est que de 20 octets (chapitre B, §1.3.3.4). Cette limitation nous interdit de référencer une *paramètre* par son nom symbolique. Il va donc falloir créer un identificateur pour chaque *paramètre*. Nous avons vu dans le paragraphe 2.6.3, qu'un *paramètre* est modélisé par une structure de données appelée ***Parametre***. Cette structure est allouée dynamiquement et son adresse jouera le rôle d'identificateur. Cela a pour avantage de dispenser le client X de faire des recherches dans une table de correspondance.

Pour éviter que deux *paramètres* en provenance de *concentrateurs* différents aient le même identificateur, l'octet de poids faible de l'adresse IP sera recopié dans l'octet de poids fort de l'identificateur (figure C.24). L'inconvénient, très mineur, de cette méthode est la limitation à 16 Moctets (2^{24} octets) de la mémoire pour stocker tous les *paramètres*.

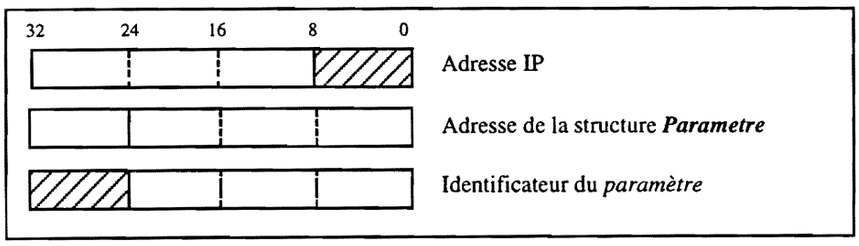


Figure C.24 - L'identificateur du paramètre

6.2.2 Représentation d'un écran GMS sur un concentrateur

Nous venons de voir que les écrans communiquent avec les *concentrateurs* par événements X. Pour pouvoir communiquer il faut associer à chaque écran un client X par concentrateur. Ce dernier sera modélisé dans le *concentrateur* par une structure de données appelée ***Client_X*** et composée par :

- *les références du serveur X (le nom et la structure de données du "display") ;*
- *l'identificateur de la fenêtre de l'écran ;*
- *l'identificateur du processus UNIX qui a lancé le client X :*
- *l'identificateur de l'utilisateur UNIX qui a lancé le client X ;*
- *le numéro de l'écran GMS ;*
- *le droits conférés à l'écran GMS (commande, mesure ou test).*

Un écran GMS sera décrit dans le *concentrateur* par un client GMS. Celui-ci est modélisé par la structure de données ***Client_GMS*** contenant :

- *la structure de données **Client_X** ;*
- *la date correspondant à la dernière émission d'une balise (§6.4) ;*
- *le nombre de défauts de la communication par événements X avec les écrans GMS (§6.2.4) ;*
- *le nombre de paramètres (mesures ou commandes graphiques) ;*
- *le tableau des identificateurs des paramètres.*

6.2.3 La liste des écrans GMS

VxWorks dispose d'une bibliothèque de fonctions pour la création et le traitement de listes doublement chaînées [22]. Chaque cellule de la liste est formée par un nœud associé à n'importe quelle structure de variable. Un nœud permet de pointer vers la cellule suivante ou la cellule précédente de la liste. Une liste doublement chaînée est initialisée par une structure contenant un pointeur sur la première cellule et un pointeur sur la dernière cellule, ainsi que le nombre de cellules dans la liste (figure C.25).

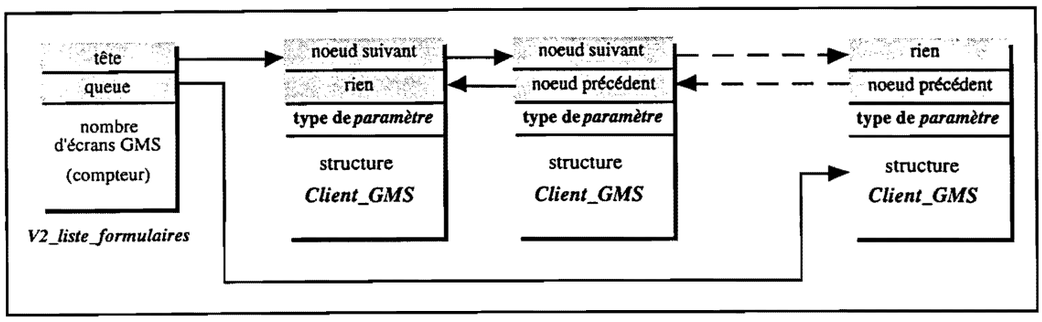


Figure C.25 - Chaînage de la liste des écrans GMS

L'ensemble des écrans connectés sur un même *concentrateur* sont groupés dans une liste doublement chaînée appelée **V2_liste_formulaires**. La figure C.24 montre comment les différentes cellules sont reliées dans la liste doublement chaînée. Chaque cellule de la liste est représentée par la structure de données ***Formulaire_GMS***, définie ci-après :

```
typedef struct {
        NODE        nœud ;        /* nœud suivant ou nœud précédent          */
        int         type ;        /* type de paramètre (commande ou mesure)  */
        Client_GMS  client_GMS ;  /* structure Client_GMS                    */
        } Formulaire_GMS ;
```

Une cellule sera insérée ou supprimée dans *V2_liste_formulaires* par les *formulaires* (§6.3) au moment de l'ouverture ou de la fermeture d'un écran sur le concentrateur. Le libre accès à la liste peut être dangereux. En effet, si une cellule est supprimée de la liste au même moment où elle est consultée, on risque de vouloir se connecter à un écran qui n'existe plus. L'accès à cette liste sera, donc, protégé par un *sémaphore d'exclusion mutuelle bloquante*.

Dans un écran GMS, les mesures représentent en moyenne 80% des *paramètres*, et les commandes graphiques les 20% restant. Ce déséquilibre peut rallonger considérablement le temps de recherche d'une commande graphique dans *V2_liste_formulaires*. Temps pendant lequel les mesures ne sont pas remontées vers les écrans de contrôle.

D'autre part, le flux de remontée des mesures est beaucoup plus important que celui des commandes graphiques. L'opérateur ne lance une commande que ponctuellement ou pendant un laps de temps déterminé (lors des réglages) alors que les mesures sont remontées en permanence par la boucle d'acquisition (§3.1). Nous pouvons considérer que 95% des échanges d'événements X correspondent à des mesures. Les clients X sont, donc, constamment occupés à remonter des mesures et un éventuel blocage de celles-ci bloquerait la connexion X et par conséquent la remontée des commandes graphiques.

Pour optimiser le temps de recherche des commandes graphiques dans *V2_liste_formulaires* et pour ne pas perturber leur remontée lors d'un éventuel dysfonctionnement des mesures, nous avons décider de dissocier les mesures des commandes graphiques. Ainsi, lors de la connexion d'un écran sur un concentrateur, les *formulaires* (§6.3) lanceront un client X pour les mesures et un client X pour les commandes graphiques. Le champ *type* de la structure ***Formulaire_GMS*** permet de différencier les mesures des commandes graphiques et d'éviter, ainsi, le parcours inutile des mesures lorsqu'on désire remonter une simple commande graphique.

6.2.4 Remontée des *paramètres*

Nous avons vu dans le paragraphe 1.3.3.4 que l'espace réservé au transport de données dans un événement X, n'est que de 20 octets. Ces octets sont répartis dans une structure de données de type ***Ev_Parametre*** :

- *1 octet pour authentifier le type de message (mesure ou commande graphique) ;*
- *1 octet pour l'émetteur (l'octet de poids faible de l'adresse IP) ;*
- *1 octet pour le numéro d'écran GMS ;*
- *1 octet pour l'état de l'alarme de la valeur (incident, alarme ou dépassement) ;*
- *4 octets pour l'identificateur du paramètre ;*
- *8 octets pour la valeur fraîche physique du paramètre ;*
- *4 octets pour la date, en secondes, de cette valeur.*

La remontée des *paramètres* est effectuée par événements X au sein de la fonction *mesureEcrans()*. (figure C.26a). La liste des clients GMS est parcourue entièrement et pour chaque client GMS la fonction *mesureEcran()* (figure C.26b) remonte toutes les mesures de l'écran. Cependant, il doit s'écouler un temps minimum entre deux remontées successives de la même mesure. Ce temps est défini par la constante **PARAM_REMONTE_DELAY** (100 ms) et analysé par la fonction *mesureAccesOn()* (§3.1).

La connexion X est ouverte lors de la réception du formulaire ('§6.3.1) et ne se referme que lorsqu'il y a trois erreurs successives de communication X (voir **IOErrorHandler** §6.4.1) ou que l'écran GMS ne le demande. Il y a deux méthodes pour envoyer un événement X : soit il est envoyé directement au serveur X soit il est regroupé avec d'autres événements dans une zone tampon qui est émise vers le serveur X par une fonction de synchronisation appartenant à la Xlib (**XSync()** (chapitre B, §1.3.3.3)). La première méthode est pratique quand les événements sont envoyés ponctuellement car ils ne doivent pas attendre l'arrivée d'autres événements. Mais le serveur X ne traite pas immédiatement l'événement ce qui fausse la dynamique de l'affichage lors de l'envoie d'un grand nombre d'événements. La synchronisation permet la prise en charge immédiate d'un événement mais coûte chère en temps. Dans notre application les événements de type **EV_MESURE** sont tous regroupés, c'est pourquoi nous utiliserons la synchronisation ce qui nous donnera une meilleure dynamique lors de l'affichage des mesures sur les écrans de contrôle.

Un problème se pose cependant lors de la communication par événements X. Il se peut qu'il y ait un problème de communication à un moment ou à un autre (connexion X coupée, réseau coupé, etc.). L'exécution du module *mesureEcrans()* peut donc rester bloquée sur un délai de connexion X (60 secondes). Pendant ce temps, aucun *paramètre* ne peut être remonté vers les écrans GMS et l'opérateur perd toute information sur la machine. La structure ***Client_GMS*** (§6.2.2) contient un

compteur de défauts, *nb_com_dfl*, indiquant le nombre de défauts de communication liés au client X de l'écran GMS. Dès la détection d'un défaut, le module *mesureEcrans()* va incrémenter ce compteur, et tant qu'il ne sera pas remis à 0 il n'y aura plus de communication avec le client X affecté.

Une tâche envoie périodiquement une **balise de mesures** à tous les clients GMS (figure C.30a, §6.4) et remet le compteur de défaut à 0 (figure C.30b) si la communication est correcte ; sinon au bout de trois défauts de communication elle retire le client GMS de la liste des clients. Ce système évite de bloquer la remontée des mesures vers les écrans de contrôle et élimine, au passage, les clients GMS qui sont défaillants.

```
mesureEcrans()

Début
    Activation du sémaphore d'exclusion mutuelle bloquant (semTake()) (§6.2.3);
    Pour chaque cellule de la liste de clients GMS faire
        Si le paramètre est de type mesure alors
            nb_com_dfl = 0;
            Remonter toutes les mesures associées à l'écran GMS (mesureEcran());
            Si défaut dans la communication X alors
                nb_com_dfl = nb_com_dfl + 1;
            FinSi
        FinSi
    FinPour
    Désactivation du sémaphore (semGive()) (§6.2.3);
Début
```

Figure C.26a - Algorithme de la remontée des mesures de tous les écrans

```
mesureEcran()

Début
    Pour toutes les mesures de l'écran GMS faire
        Récupération de l'adresse du paramètre à partir de son identificateur;
        Si la remontée du paramètre est autorisée alors
            Préparation des 20 octets de données pour l'événement X;
            Envoi de l'événement X en mode synchronisation (chapitre B, §1.3.3.3);
            Incrémenter le compteur de mesures envoyées, V2_mesures_cpt (variable globale);
        FinSi
    FinPour
    Envoie groupé de tous les événements X (XSync());
Début
```

Figure C.26b - Algorithme de la remontée des mesures d'un écran

La remontée d'une commande graphique est effectuée sur demande par la *fonction de commande*. Le principe (figure C.26c) diffère légèrement de celui des mesures. Le module *parametreRemonteCommandeGfx()* parcourt la liste des clients GMS et pour chaque écran il va identifier la commande à remonter. S'il y a un défaut de communication X, le compteur de défaut est incrémenté. Par contre, la communication sera réessayée tant que la tâche des **balises de commandes** (§6.4) n'aura pas supprimé l'écran défaillant de la liste des clients GMS. Ceci est justifié par le fait qu'on ne peut pas ignorer un client qui vient de faire une commande et qui ne réponds plus. De plus, la remontée d'une commande est ponctuelle et la recherche des commandes dans la liste est séparée de la recherche des mesures. Le risque de bloquer la remontée des commandes graphiques vers les écrans de contrôle est donc très faible.

```
parametreRemonteCommandeGfx()

Début
    Préparation des 20 octets de données;
    Activation du sémaphore d'exclusion mutuelle bloquant (semTake()) (§6.2.3);
    Pour chaque cellule de la liste de clients GMS faire
        Si typede paramètre = CLIENT_GMS_COMMANDE alors
            Pour toutes les commandes faire
                Si commande = commande recherchée alors
                    Récupération du numéro de l'écran GMS;
                    Envoie de l'événement X (clientXEnvoi());
                    Si défaut dans la communication X alors
                        nb_com_dfl = nb_com_dfl +1;
                    FinSi
                FinSi
            FinPour
        FinSi
    FinPour
    Désactivation du sémaphore d'exclusion mutuelle bloquant (semGive()) (§6.2.3);
Fin
```

Figure C.26c - Algorithme de la remontée des commandes graphiques

6.3 Les formulaires

Pour initier les communications X il est nécessaire d'informer chaque client de l'identité de ses correspondants. Cela ne peut évidemment être fait par des événements X. Il faut donc passer par un service de communication propre. Ce rôle est rempli par un **serveur de formulaires**, utilisant le support de communication des RPC en mode TCP, pour rester homogène avec ce qui a été fait.

Le **formulaire** est le média utilisé par les différents acteurs de contrôle et commande pour s'échanger des informations sur des *paramètres* lors de la connexion écran-*concentrateur* ou d'un démarrage à un état donné. Il est modélisé par une structure de données appelée ***Formulaire*** et contenant :

- *l'objet de la requête ;*
- *le nom symbolique du concentrateur ;*
- *la structure de données **Client_X** (§6.2.2) ;*
- *l'état du concentrateur ;*
- *le nombre de paramètres à échanger ;*
- *pour chaque paramètre son symbole, son identificateur et sa valeur ;*
- *un état indiquant que la communication RPC s'est bien passée.*

Le principe (figure C.27) consiste à s'échanger des *formulaires* mettant en regard pour chaque *paramètre*, un identificateur, un symbole et une valeur. En partant de l'un de ces membres et selon la nature de la requête, le client devra renseigner les autres champs de la structure. L'objet de la requête permet au *concentrateur* de savoir s'il doit lancer ou détruire un client X ou s'il doit échanger des renseignements liés aux *paramètres*. La requête est constituée d'un mot de 32 bits répartis comme le montre la figure C.28. Une requête est construite avec l'opérateur logique **OU**. Donnons pour exemple la requête correspondante à l'ouverture d'un écran GMS : **FROM_SYMB | TO_ID | START_CLIENT**. L'application GMS demande au concentrateur de lui fournir l'identificateur (TO_ID) associé à son symbole (**FROM_SYMB**) et de lancer un client X (**START_CLIENT**).

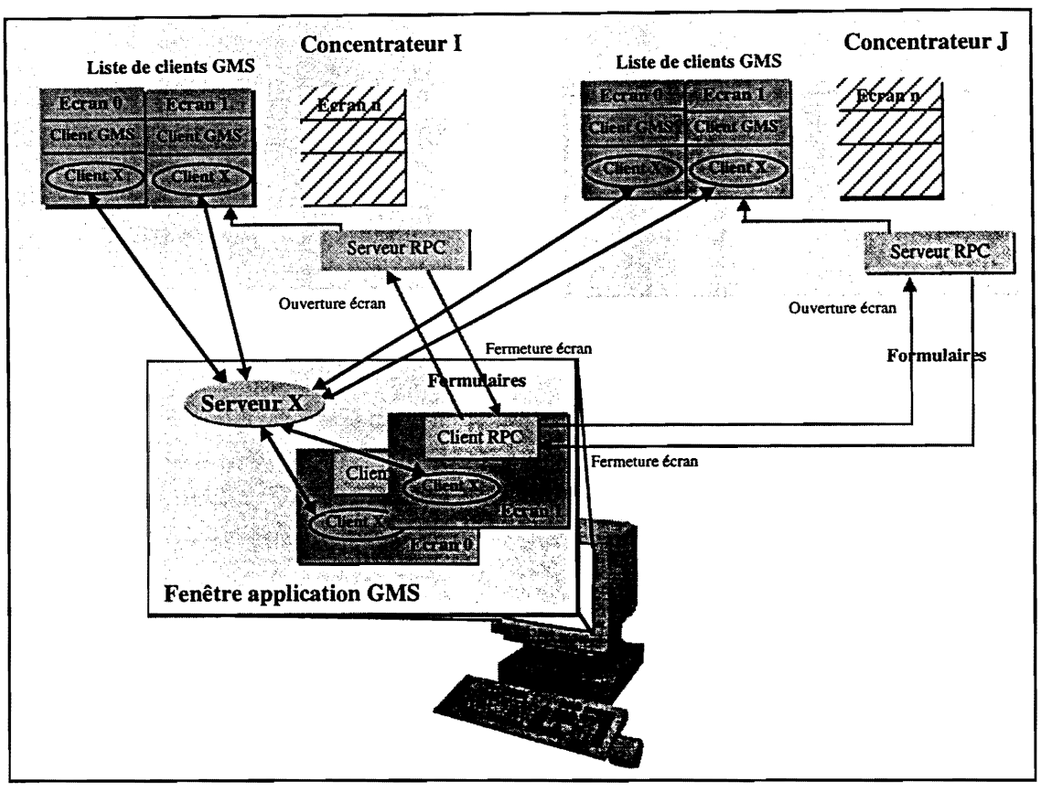


Figure C.27- Principe de la remontée des commandes graphiques

La valeur du *paramètre* permet éventuellement au concentrateur, de retourner les dernières consignes à l'écran en cas de reprise ou de changement de poste de contrôle. A partir de ces consignes, l'écran se fera une remise à jour graphique sans exécuter la moindre commande.

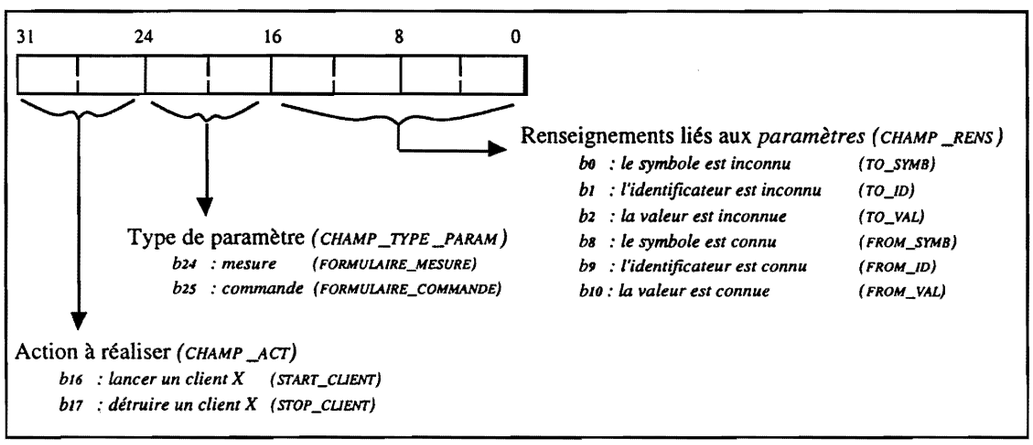


Figure C.28 - Composition du champ requête de la structure Formulaire

6.3.1 Ouverture d'un écran GMS

Lors de l'ouverture d'un écran, l'application GMS ne connaît qu'une liste de symboles, non identifiés, associées avec des éléments graphiques. De son côté, le *concentrateur* connaît les identificateurs de ces symboles mais ne peut pas les véhiculer car la communication X n'est pas établie. Le serveur de *formulaires* servira à établir cette connexion et à associer à chaque symbole son identificateur et sa valeur :

- L'écran (client RPC) émet à chaque *concentrateur* un premier *formulaire* contenant la liste des symboles des mesures et des commandes et demandant la création d'un client X (action = **START_CLIENT**). Ceci est fait par la fonction *ecranStructFormulairesOn()* exécutée dans la méthode *activate* de l'écran GMS (§6.1.4.2).
- Le serveur RPC de *formulaires* établit la connexion X (il vérifie avant que la connexion n'est pas déjà été créé) et crée en mémoire un client GMS qui s'ajoute à la liste des clients GMS (figure C.29b). Le *formulaire*, rempli avec les identificateurs et les valeurs de chaque mesure, est retourné via les RPC à l'écran GMS (voir fonction *StartClient()*, figure C.29a).
- GMS prévient *xvivetat* (§8).

```
StartClient()

Début
    Remplissage de la structure Client_GMS à partir des informations fournies par le message RPC
    (clientGMSFormulTransfert());
    Recherche du type de paramètre sur le champ requête du formulaire;
    Exécution de la fonction clientGMSDansListe();
    Remplissage de la structure Formulaire à retourner à l'écran GMS client;
Fin
```

Figure C.29a - Algorithme de création d'un client GMS

```
clientGMSDansListe()

Début
    Allocation de la mémoire pour la structure Formulaire_GMS;
    Remplissage de la structure;
    Si le client GMS existe déjà
        Alors Libérer la mémoire de la structure Formulaire_GMS;
              Retourner un message d'erreur;
        Sinon Création de la connexion X;
              Si la communication n'est pas défaillante alors
                  Insertion du client GMS dans la liste des clients GMS;
              FinSi
    FinSi
Fin
```

Figure C.29b - Algorithme d'insertion d'un client dans la liste des clients GMS

6.3.3 Mise à jour d'un écran GMS

Une fois que l'écran GMS est ouvert et que les connexions X sont établies avec les *concentrateurs*, il se peut que la connexion écran-*concentrateur* soit coupée momentanément par un acteur extérieur (coupure réseau, *concentrateur* défectueux, etc.). Cet état va être détecté par l'écran

GMS qui ne va pas se refermer mais qui va essayer de se reconnecter régulièrement au(x) concentrateur(s) défaillant(s) (voir méthode *update*, paragraphe 6.1.4.3). Dès que la connexion est à nouveau rétablie un *formulaire* est envoyé au concentrateur, par la fonction *concFormulairesEchange()* de la méthode *update* (§6.1.4.3). Ce *formulaire* est identique à celui envoyé lors d'une ouverture d'écran, seulement, si le client GMS existe toujours dans le *concentrateur* il ne sera pas créé une nouvelle fois.

6.3.2 Fermeture d'un écran GMS

Lors de la fermeture d'un écran, l'application GMS doit le signaler aux *concentrateurs* concernés pour que les clients GMS soient supprimés de la liste de clients GMS de chaque concentrateur. Cette action est effectuée par la fonction *ecranStructFormulairesOff()* exécutée dans la méthode *deactivate* de l'écran GMS (§6.1.4.4) :

- GMS émet un premier *formulaire* demandant la fermeture du client X associé aux mesures (action = STOP_CLIENT).
- Le *concentrateur* ferme la connexion X et répond si ça c'est bien passé ou non.
- GMS émet un deuxième *formulaire* demandant la fermeture du client X associé aux commandes.
- Le *concentrateur* ferme la connexion X et répond si ça c'est bien passé ou non.
- GMS prévient *xvivetat* (§8).

6.4 Les balises

Une fois le système initialisé, l'écran de contrôle doit pouvoir gérer l'absence d'un concentrateur pour connaître l'état réel du système. Le concentrateur va se manifester suivant une certaine périodicité par le biais de **balises**. Les *balises* vont permettre à l'écran GMS d'identifier le(s) concentrateur(s) et de connaître périodiquement ces comportements (les différents états, le nombre de boucles effectuées par la tâche d'acquisition *tPollAcq*, etc.).

Une *balise* est un événement X de type **EV_BALISE** qui va être inséré dans la boucle de traitement des événements X de GMS (figure C.20). L'espace réservé au transport de données dans un événement X est de 20 octets (§1.3.3.4) qui sont répartis dans une structure de données de type ***Ev_Balise*** de la manière suivante :

- *1 octet pour authentifier le type de message (balise) ;*
- *7 octets pour le nom du concentrateur(plus 1 octet de fin de chaîne de caractère non utilisé dans l'événement) ;*
- *1 octet pour l'état du concentrateur ;*
- *4 octets pour l'adresse IP du concentrateur (chapitre B, §1.2.2.1) ;*
- *4 octets pour le nombre de boucles de tPollAcq ;*
- *3 octets pour le terminateur de l'événement ('\0', '\0', '\0').*

Si un *concentrateur* ne remonte plus de mesures on ne peut rien conclure : la tâche d'acquisition peut être momentanément suspendue (le compteur d'acquisition ne s'incrémente plus). Par contre, si nous

ne recevons plus de *balises* c'est que le *concentrateur* est planté. L'écran va alors essayer de se reconnecter périodiquement. Cet état ne nuit en rien aux fonctions dépendant des autres *concentrateurs*.

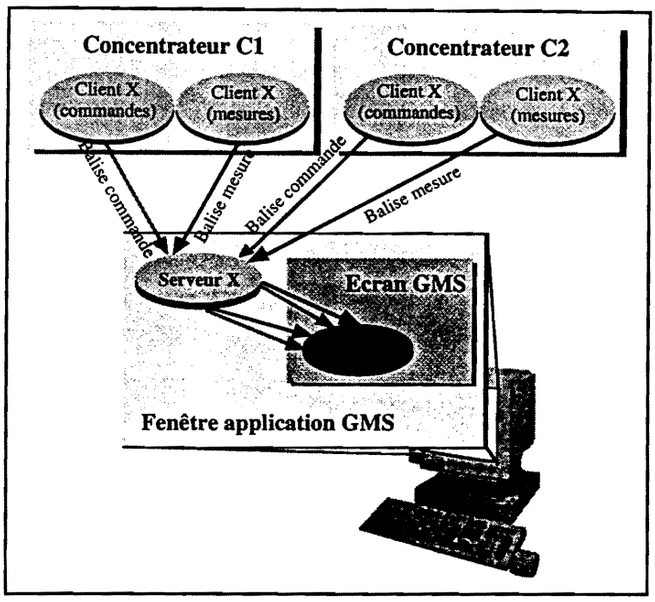


Figure C.30a - Principe des balises

6.4.1 Traitement au niveau des concentrateurs

```
appliStructVerifCommunication()

Début
    verif_compt = 0;  /* compte le nombre de vérifications de connexion (§6.1.2) */
    Pour  tous les concentrateurs de la liste des concentrateurs de l'application GMS  faire
        Si  le concentrateur est utilisé par l'écran GMS  alors
            verif_compt = verif_compt + 1;
            Si  le temps d'arrivée de la balise > 20 secondes  alors
                La balise est prise en compte;
                Si  le compteur de tPollAcq ne s'incrémente plus  alors
                    Positionner le champ "état" du concentrateur (§6.1.2);
                FinSi
                Si  le temps entre la balise précédente et la nouvelle ≥ 40 secondes
                    Alors  L'écart est trop important;
                        Si  verif_compt ≥ 10  alors
                            Le concentrateur n'émet plus de balises;
                            Insérer le concentrateur dans la liste des concentrateurs défectueux (§6.1.4.1);
                        FinSi
                    Sinon verif_compt = 0;
                FinSi
            FinSi
        FinSi
    FinPour
    Retourner le nombre de concentrateurs défectueux;
Fin
```

Figure C.30b - Algorithme de l'émetteur de balises

Une tâche, appelée **tBalise**, envoie régulièrement (toutes les 5 secondes) une balise à chaque écran GMS concerné. Pour cela, elle parcourt la liste de clients GMS dont l'accès est protégé par un sémaphore d'exclusion mutuelle bloquante (§6.2.3), et pour chaque client GMS elle envoie un événement X de type **EV_BALISE**. Si lors de cet envoie survient une erreur de communication X, c'est que l'un des clients X composant la connexion (côté écran ou côté concentrateur) est "mort". La connexion X doit être à nouveau refaite ce qui implique le lancement d'un nouveau client X et donc d'un nouveau client GMS, côté concentrateur. L'ancien client GMS doit donc être supprimé de la liste avant de créer le nouveau sous peine de conflit (le numéro de fenêtre associé à l'écran GMS a changé...).

La suppression du client GMS est gérée par le gestionnaire d'erreurs, **IOErrorHandler**, auquel nous avons associé une fonction utilisateur (chapitre B, §1.3.3.4). Le client GMS affecté est transmis dans une variable globale au *IOErrorHandler* (*client_GMS_IOError*) qui, avant de couper la connexion X, va supprimer le client de la liste de clients GMS et relancer une nouvelle tâche *tBalise* (l'ancienne disparaît lors de l'erreur d'Entrée/Sortie).

6.4.2 Traitement au niveau de l'application GMS

Le traitement des balises au niveau de l'application GMS s'effectue dans la fonction *appliStructVerifCommunication()* exécutée par la méthode *update* de chaque objet écran (figure C.23c). Cette fonction va retourner la liste des *concentrateurs*, *conc_en_defaut* (§6.1.4.1), qui ont une communication défectueuse avec l'application GMS (figure C.30c).

```
appliStructVerifCommunication()

Début
    verif_compt = 0;  /* compte le nombre de vérifications de connexion (§6.1.2) */
    Pour tous les concentrateurs de la liste de concentrateurs de l'application GMS faire
        Si le concentrateur est utilisé par l'écran GMS alors
            verif_compt = verif_compt + 1;
            Si le temps d'arrivée de la balise > 20 secondes alors
                La balise est prise en compte;
                Si le compteur de tPollAcq ne s'incrémente plus alors
                positionner le champ "état" du concentrateur (§6.1.2);
                FinSi
                Si le temps entre la balise précédente et la nouvelle ≥ 40 secondes
                    Alors L'écart est trop important;
                        Si verif_compt ≥ 10 alors
                            Le concentrateur n'émet plus de balises;
                            Insérer le concentrateur dans la liste des concentrateurs défectueux (§6.1.4.1);
                        FinSi
                    Sinon verif_compt = 0;
                FinSi
            FinSi
        FinSi
    FinPour
    Retourner le nombre de concentrateurs défectueux;
Fin
```

Figure C.30c - Principe du traitement des balises dans une application GMS

La fonction *appliStructVerifCommunication()* vérifie, tout d'abord, que 20 secondes se soient écoulées depuis l'arrivée de la dernière balise (rappelons que le *concentrateur* envoie une balise toutes les 5

secondes). Si c'est le cas elle prend en compte la balise et l'analyse. Si après 50 secondes (40s + 10s) (figure C.30c) l'écran n'a pas reçu de balise le *concentrateur* est déclaré défectueux et insérer dans la liste d'écrans défectueux.

7 Représentation du système de contrôle et commande dans le SGBD O2

Par rapport à la première architecture où la base de données prenait essentiellement en charge l'aspect matériel du système de contrôle et commande, dans la nouvelle architecture la base prend en charge, en plus de l'aspect matériel (concentrateur), l'aspect fonctionnel du système (fonctions de mesure et de commande, calibration, comportement graphique des écrans GMS, etc.) et la gestion des écrans GMS. Ces informations sont très souvent utilisées dans le code des programmes devant gérer le système de contrôle et commande. Donc, pour éviter une fastidieuse réécriture du même code lors de la création d'un nouveau paramètre, nous avons décider de donner à la base O_2 le rôle de générer des modules de code. De cette façon, nous intégrons dans la base, en plus de la description des équipements, le code des programmes devant les gérer. Nous arrivons, ici, à l'aspect le plus original de cette thèse, la génération de code par une base de données. Cette fonctionnalité n'est pas habituelle dans un système de gestion de base de données qui a surtout le rôle de décrire, stocker, accéder et maintenir de grosses quantités de données.

Cette nouvelle utilisation de la base se traduit par la création de trois **racines de persistance**, déclarées dans la base par trois **objets nommés** :

- **liste_des_concentrateurs**
  pour l'aspect matériel, regroupant les objets de la classe ***Concentrateur*** ;
- **liste_des_équipements_atomiques**
  pour les fonctionnalités, regroupant les objets de la classe ***Equipement_Atomique*** ;
- **liste_des_applications**
  pour les écrans GMS, regroupant les objets de la classe ***Appli_GMS***.

Grady Booch [37] définit la persistance comme étant « *la propriété qui permet à un objet de continuer d'exister après que son créateur ait cessé d'exister, y compris à un endroit différent de celui où il a été créé* ». Notons que dans le Système de Gestion de Bases de Données O_2, la persistance est associée à la notion de **racines de persistance** et s'applique aussi bien aux objets qu'aux attributs (chapitre B, §3.3.2.1). Un objet devient persistant s'il est attaché directement ou indirectement (via un objet déjà persistant) à une racine de persistance. Précisons qu'un "lien d'hérìtage" entraîne une relation d'appartenance entre deux objets, l'objet fils héritant des attributs et des méthodes de l'objet père ; tandis qu'un **lien logique** permet simplement l'accès à un objet B depuis un objet A, A et B n'ayant aucune relation d'appartenance.

Dans O_2 un objet persistant est détruit à la fermeture de la session O_2 s'il n'appartient plus directement ou indirectement à une racine de persistance. Prenons le premier exemple de la figure C.31. L'objet B appartient à la même racine de persistance que l'objet A, et l'objet C appartient à une autre racine de persistance. La destruction de A devrait entraîner naturellement la destruction de tous les objets qui le composent, c'est à dire B. Ors B est attaché indirectement à C et donc, il ne sera pas détruit. Cette propriété n'est pas sans conséquences pour la cohérence des données de la base. Citons l'exemple des mesures : un équipement est composé d'un certain nombre de mesures, attachées à un *concentrateur* donné. Si un équipement est détruit, ses mesures restent dans la base car elles sont attachées au concentrateur. Nous contournerons cette difficulté en créant des doubles liens entre les objets et en n'autorisant l'accès aux objets à détruire, qu'à travers des méthodes. Prenons le deuxième exemple de la figure C.31. Les objets B et C ont maintenant un double lien. L'objet A, avant d'être détruit, va déclencher une méthode qui via les liens de persistance, pourra accéder à l'objet C pour couper le

lien C --> B. Ainsi B sera libéré de la racine de persistance de C, et sera détruit avec A.

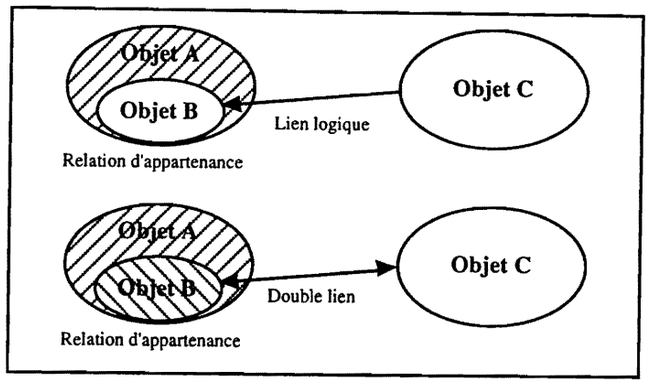


Figure C.31 - Exemples d'objets persistants

La notion de persistance dans O_2 étant précisée, nous allons introduire deux concepts importants pour présenter ensuite une vue globale du système de contrôle et commande dans la base.

7.1 Les équipements

Les équipements du projet Vivitron peuvent être décomposés en deux catégories : les équipements de base, où sont regroupées les mesures et les commandes, et les équipements composés de plusieurs sous-éléments. Le concept d'**équipement atomique** sera attaché à la première catégorie et le concept d'**équipement composite** à la deuxième catégorie. Toute fois, un *équipement composite* peut être constitué d'un ou plusieurs *équipements composites* et/ou d'un ou plusieurs *équipements atomiques*. Nous illustrons ces deux concepts dans la figure C.32 où l'injecteur (§1.3.2) a été pris comme exemple.

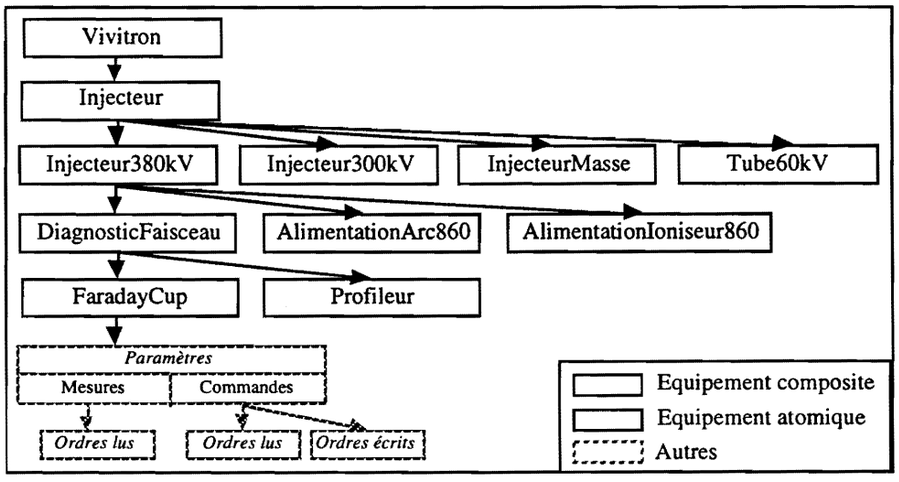


Figure C.32 - Les équipements composants l'injecteur du Vivitron et leurs liens d'appartenance

7.2 Représentation globale

Le système de contrôle et commande vu du côté de la base est décomposé en trois parties : la description du matériel, la description des fonctionnalités et la description des écrans GMS. Chaque partie est représentée par une racine de persistance dont les principaux objets apparaissent sur la figure C.33.

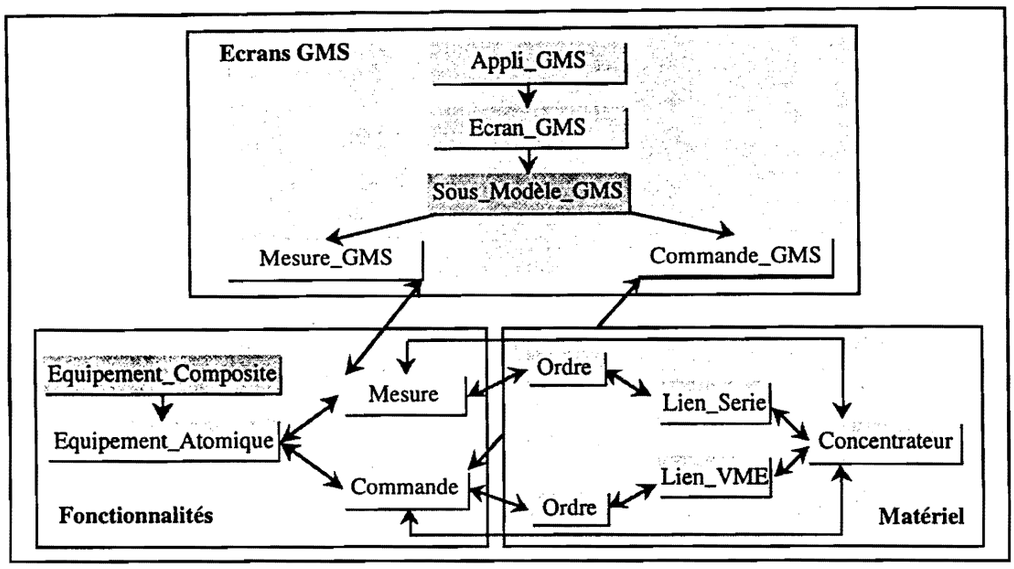


Figure C.33 - Graphe de persistance du système de contrôle et commande

Les paragraphes suivants vont nous décrire plus en détail les composants de chaque racine de persistance.

7.3 Description du matériel

La description de la partie matérielle est stockée dans la base suivant un schéma d'objets se rapprochant le plus possible de son implantation physique. Il s'agit de *concentrateurs* composés de cartes d'interface (cartes VME et/ou automates (§5)) contenant un ensemble de canaux. L'information va circuler depuis les *concentrateurs* jusqu'aux canaux pour agir sur les capteurs et les actionneurs. Cette représentation doit se retrouver naturellement dans la base de données (figure C.33).

La figure C.34a nous montre comment tous les éléments matériels héritent de la superclasse ***Reference***. Elle sera utilisée par ses sousclasses pour accéder : au nom de l'équipement matériel ; à une image, une vue graphique ou une description du matériel ; à la référence du fournisseur.

Rappelons qu'un *concentrateur* (chapitre B, §4.1.1) est composé d'une carte processeur gérant des cartes d'interface (*liens VME*) logées dans un même châssis VME. Ces cartes, reliées au bus VME sont accessibles par une adresse VME. Cet aspect est reflété dans la base par les sousclasses ***Concentrateur*** et ***Lien_VME***, héritières de la classe ***Carte_VME*** qui contient l'adresse VME de toute

carte VME (figure C.34a). Notons que ce sont les objets de la classe ***Concentrateur*** qui vont regrouper toutes les fonctions C qui gèrent le concentrateur.

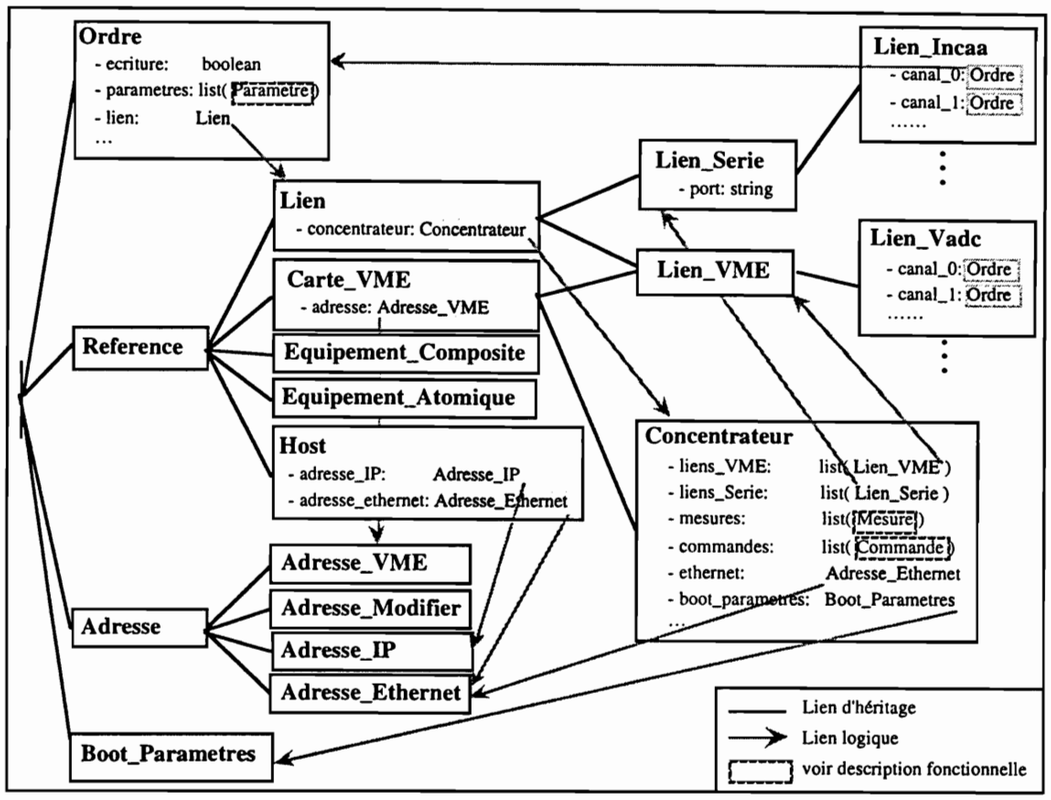


Figure C.34a - Le graphe d'héritage de la partie matérielle

Au niveau de la modélisation, il n'y a pas de différence entre un *lien série* et un *lien VME*, si ce n'est que le premier est référencé par son port série et le deuxième par son adresse VME. Tous les deux sont modélisés par les sousclasses ***Lien_Serie*** et ***Lien_VME*** qui héritent de la classe ***Lien*** dont le seul rôle et de référencer le *concentrateur* auquel appartient le *lien*. Chaque *lien* possède un certain nombre de canaux pour accéder aux capteurs ou aux actionneurs, qui varie selon le type de carte. Pour chaque type, nous avons créé une sousclasse héritière des classes ***Lien_Serie*** ou ***Lien_VME***, avec des attributs qui modélisent tous les canaux de la carte (objets de la classe ***Ordre***). Les méthodes de ces sousclasses (*init*, *edit*, *liste_ordres*, ...) ont la même fonctionnalité d'une sousclasse à l'autre, leur contenu diffère suivant la liste d'*ordres* associée à chaque type de *lien*. Pour éviter une fastidieuse réécriture de toutes les méthodes chaque fois qu'un nouveau type de *lien* est créé, nous avons mis au point un **générateur de *liens*** qui crée automatiquement la classe et les méthodes du *lien*.

Dans le chapitre 2.3 nous avons vu que la lecture et l'écriture sur un canal sont associées à un *ordre* aussi, dans la base, la description physique d'un canal va être modélisée par la classe ***Ordre***. Cette classe est composée notamment de la liste des *paramètres* qui utilisent l'*ordre* et du *lien* auquel l'*ordre* appartient.

Les *concentrateurs* communiquent avec les cartes VME via un bus VME, et avec les écrans graphiques via un réseau Ethernet. Pour s'identifier entre eux ils ont besoin d'adresses (adresse VME ou adresse Ethernet). Celles-ci sont modélisées par la classe ***Adresse***. qui contient les différents types

d'adresses auxquels vont faire appel les *liens VME*, les *concentrateurs* ou les machines hôtes (identifiées par la classe ***Host***). Pour démarrer, un *concentrateur* utilise des paramètres de démarrage (paramètres de *boot*) qui peuvent être modifiés suivant la configuration choisie par l'utilisateur. Ces paramètres vont être stockés dans les objets de la classe ***Boot_Parametres***.

La description matérielle du système de contrôle et commande est finalement axée sur les objets de la classe ***Concentrateur*** car ce sont eux qui vont former la racine de persistance ***liste_des_concentrateurs***. Nous n'avons pas parler dans ce paragraphe des *équipements composites* ou *atomiques*. En effet, ils peuvent représenter des équipements physiques mais leur rôle est plutôt de référencer les *paramètres* de ces équipements, les mesures et les commandes, qui eux relèvent de la fonctionnalité. C'est pourquoi, nous pouvons voir les sousclasses ***Equipement_Composite*** et ***Equipement_Atomique*** attachées à la superclasse ***Reference***, pour le côté matériel, mais contenant des attributs dont leur rôle a un aspect plutôt fonctionnel (figure C.34b).

7.4 Description des fonctionnalités

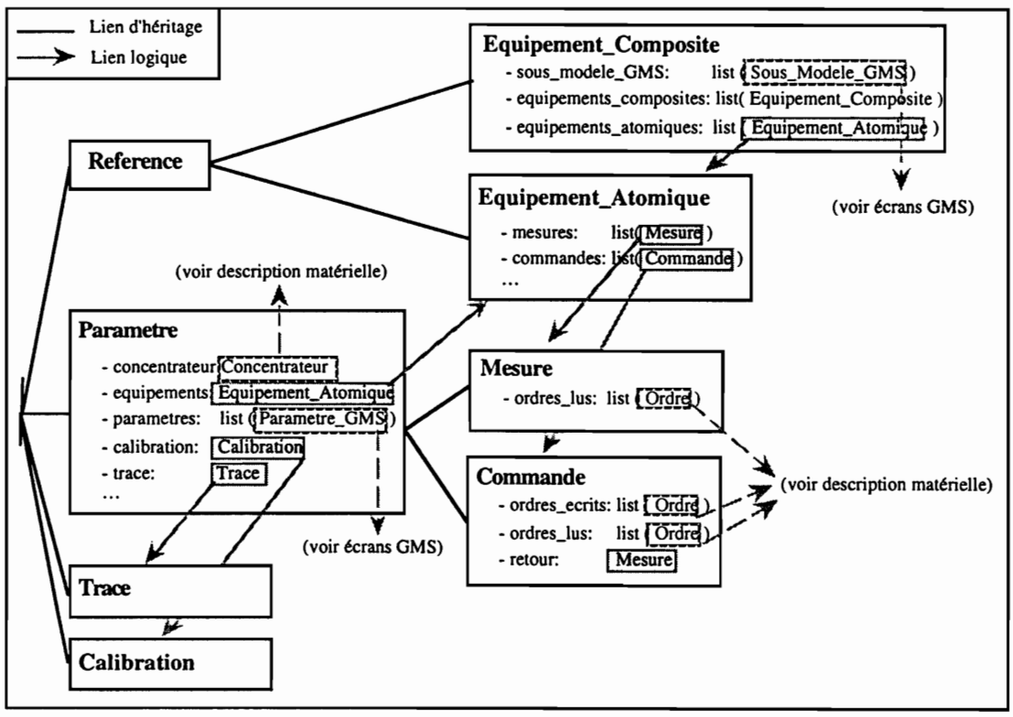


Figure C.34b - Le graphe d'héritage de la partie fonctionnelle

Dans le paragraphe 2.3 nous avons vu la notion de *paramètre* et la notion d'*ordre*. Ces deux notions sont liées, un *paramètre* étant composé d'une liste d'*ordres*, mais l'*ordre* modélise la description matérielle d'un canal tandis que le *paramètre* modélise le fonctionnement d'un ensemble de canaux. Nous pouvons donc considérer que les fonctionnalités sont regroupées au sein des équipements sous forme de mesures et commandes (*paramètres*), intégrant leur fonction associée (§3.3, §4.1), leur paramètres de calibration, leur *trace* (§10) et leur comportement graphique. Cette

description est reflétée dans la base, où la classe ***Equipement_Atomique*** est liée aux mesures et aux commandes par deux listes d'objets des classe ***Mesure*** et ***Commande***. Ces deux classes sont deux sousclasses de la superclasse ***Parametre*** où sont référencées toutes les fonctionnalités des *paramètres* (figure C.34b). Notons que le code des fonctions associées aux mesures et aux commandes est regrouper au sein des objets des classes ***Mesure*** et ***Commande***.

Dans cette description ce sont les objets de la classe ***Equipement_Atomique*** qui vont former tout naturellement la racine de persistance ***liste_des_équipements_atomiques.***

7.5 Les écrans GMS

Une application GMS est composée d'un écran d'accueil et d'un ensemble de sous-écrans, chaque écran étant, bien sûr, un écran GMS. L'écran GMS, lui, est composé d'objets graphiques (thermomètre, bouton On/Off, voyant, etc.) formant des groupes qui représentent graphiquement une partie élémentaire du système de contrôle et commande (§6.1). Ces groupes sont appelés des **sous-modèles** et l'ensemble de *sous-modèles* formant l'écran GMS est appelé **modèle** (chapitre B, §3.4.1). Le graphe de la figure C.34c nous montre cette représentation. La classe ***Appli_GMS*** est constituée d'un objet de la classe ***Ecran_Accueil*** et d'une liste d'objets de la classe ***Ecran_GMS***. Elle regroupe les fonctions nécessaires à la création d'un programme exécutable sous UNIX. Chaque écran est doté de plusieurs méthodes propres à GMS (*update*, *activate*, *deactivate* et *deactivate_substates*) (§6.1.4). Chacune de ces méthodes est représentée comme un membre spécifique de la classe ***Ecran_GMS***, classe qui regroupe le code C de ces méthodes.

Un *sous-modèle* se construit à partir d'une description graphique extraite d'un fichier GMS et comportant des éléments de décor, de position, d'échelle et des *variables dynamiques*. Le décor est un texte qui regroupe toutes les descriptions graphiques statiques. Les ressources GMS décrivant les éléments statiques de l'écran sont modélisées par la classe ***Decor_GMS***. Sa sousclasse, ***Decor_Ecran_GMS***, ajoute les éléments actifs de l'écran : les objets de la classe ***Sous_Modele_GMS***. La classe ***Ecran_GMS*** hérite de ***Decor_Ecran_GMS*** en lui ajoutant toutes les parties relatives au code exécutable. C'est elle qui fournit le fichier complet des ressources GMS.

La description graphique dynamique d'un *sous-modèle* est gérée par les **variables dynamiques**. Rappelons qu'une *variable dynamique* GMS (chapitre B, §3.4.1) est une variable qui modifie dynamiquement les propriétés d'un objet graphique GMS (couleur, thermomètre, texte, etc.). Elle est modélisée dans la base par la classe ***Var_Dyn_GMS***. Le rôle de O_2 va être d'associer à ces *variables dynamiques* les variables du comportement graphique ainsi que les mesures et les commandes du système. Le comportement graphique des écrans GMS définit par exemple : les mêmes couleurs pour tous les objets graphiques ; le nombre de graduations ainsi que les valeurs maximales et minimales de cette graduation ; le réglage du pas d'un curseur (§6.1.1) ; l'unité ; l'état d'un bouton (clignotant ou pas). Tous ces paramètres sont stockés dans la base sous forme d'objets de la classe ***Comp_Graphique***. Ces objets sont récupérés par la superclasse ***Parametre_GMS*** pour les regrouper avec les commandes, les mesures et les *variables dynamiques* autour des sousclasses ***Mesure_GMS*** et ***Commande_GMS***.

Chaque *sous-modèle* possède un certain nombre de *variables dynamiques*, de *mesures GMS* et de *commandes GMS*, qui varie selon le type de sous-modèle. Comme pour les *liens* (§7.3), un

générateur de sous-modèles créant automatiquement la classe et les méthodes du *sous-modèle*, a été mis au point. Pour connaître le panel de *paramètres* à relier aux *variables dynamiques*, nous avons relié la classe ***Sous_Modele_GMS*** à la classe ***Equipement_Composite***.

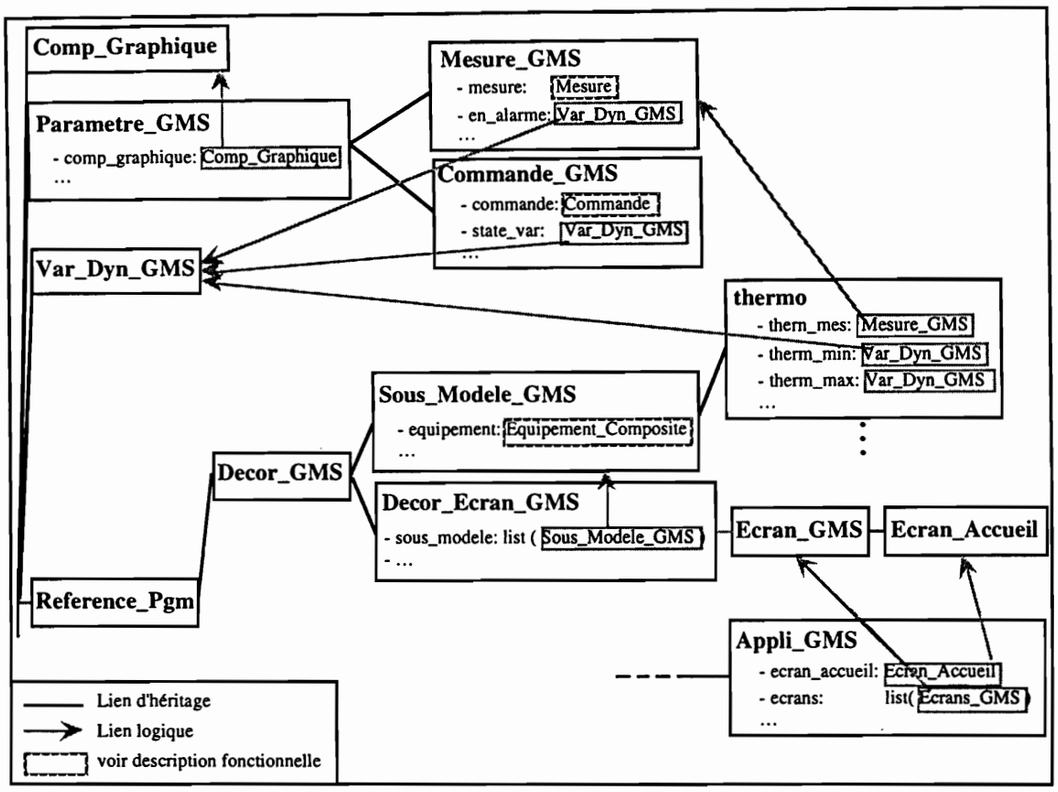


Figure C.34c - Le graphe d'héritage d'une application GMS

Enfin, nous noterons que les objets attachés aux écrans GMS vont devenir persistants par le biais de l'objet nommé ***liste_des_applications***. Cette racine de persistance est formée d'objets de la classe ***Appli_GMS*** car c'est l'application GMS qui gère l'ensemble des écrans GMS. Nous noterons, aussi, que la classe ***Decor_GMS*** hérite de la superclasse ***Reference_Pgm*** qui référence tout ce qui concerne un écran GMS depuis sa programmation jusqu'à son comportement graphique. Mais cette superclasse est liée surtout à la partie génération de code que nous abordons dans le paragraphe qui suit.

7.6 Le générateur de code

Nous présentons ici l'aspect le plus original de cette thèse : la génération de code par une base de données. Cette partie sera détaillée dans le paragraphe 11, nous ferons donc une brève introduction en présentant les classes "utilitaires" qui modélisent la gestion du code dans la base de données O_2.

La génération concerne autant les programmes appartenant aux applications GMS que les programmes s'exécutant sur les *concentrateurs*. La classe ***Fonction_C*** modélise le code source de toute fonction écrite en langage C ANSI. Elle hérite de la classe ***Text*** qui appartient à la bibliothèque de classes de

O_2 (O_2kit) et qui permet la manipulation d'un fichier texte UNIX. Les fonctions créées pour les *concentrateurs* sont des fonctions C qui demandent certaines particularités (le traitement des alias et la conservation des modules objet VxWorks (§11)). C'est pourquoi nous avons créé la classe ***Fonction_Conc*** qui hérite de ***Fonction_C***. Les applications GMS sont dotées de quatre méthodes pour gérer les écrans GMS (*update*, *activate*, *deactivate* et *deactivate_substates* (§6.1.4)). Pour chaque méthode nous avons créer une classe qui hérite de ***Fonction_C*** et qui personnalise la fonction C (nom, date, auteur, etc.). Nous avons fait de même pour le programme principal de l'application GMS en ajoutant le corps de cette fonction (classe ***Main_C***).

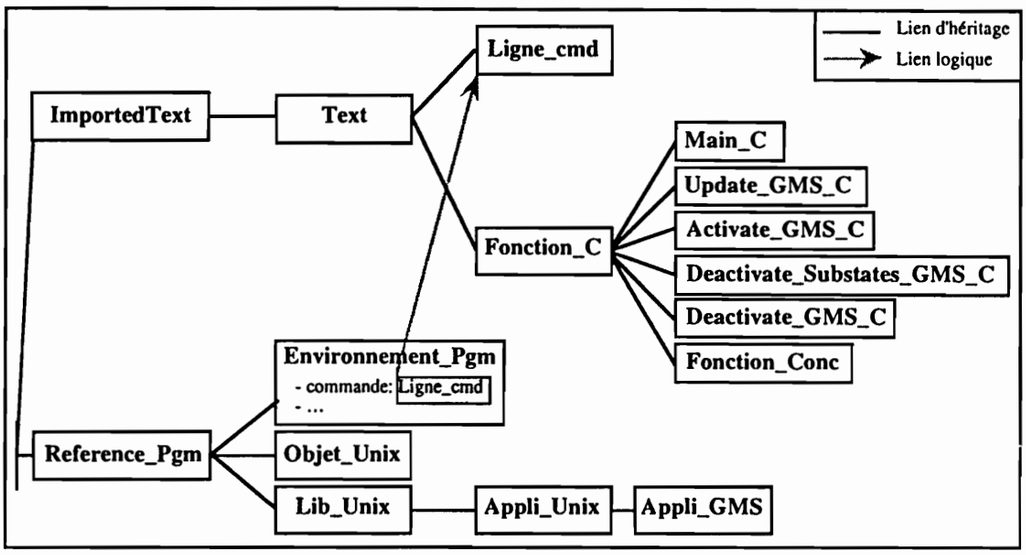


Figure C.34d - Le graphe d'héritage du générateur de code

La réalisation d'une application exécutable sous UNIX passe par deux étapes : la compilation des fonctions C dépendantes de l'application pour créer des modules objet et le regroupement de ces modules pour construire le fichier exécutable. La classe ***Objet_Unix*** récupère une liste des fonctions C et délivre le modules source ou objet UNIX correspondant. La classe ***Lib_Unix*** rassemble tous les modules objet UNIX pour créer une bibliothèque ou un fichier exécutable. La classe ***Appli_Unix*** hérite de cette dernière en ajoutant le programme principal de l'application UNIX (objet de la classe ***Main_C***). La classe ***Appli_GMS*** hérite de la classe ***Appli_Unix*** en lui ajoutant une fonction d'initialisation et une fonction de traitement des événements X (§6.1.4.1). Ce sont ses méthodes qui vont construire le fichier exécutable de l'application GMS.

Les fonctions gérées par la base de données sont éditées et compilées dans un environnement propre à UNIX pour les applications GMS, ou propre à VxWorks pour les modules des *concentrateurs*. Ces deux environnements sont définis par la classe ***Environnement_Pgm***. Les lignes de commande de compilation sont créées dans la classe ***Ligne_cmd*** et rajoutées aux objets de la classe précédente.

8 Le gestionnaire d'écrans : *xvivetat*

Un écran GMS peut être ouvert une ou plusieurs fois sur la machine locale ou sur une machine distante. Cet aspect a pour avantage de pouvoir contrôler les mêmes *paramètres* sur des sites différents. Par exemple, un premier écran injecteur pourra être ouvert dans le local de l'injecteur pour régler l'injection du faisceau sur place et un deuxième écran pourra être ouvert dans la salle du pupitre pour suivre les réglages. Cependant, un problème se pose quand un même écran est ouvert plusieurs fois : il peut être dangereux de commander plusieurs écrans en même temps. De même, une fois les réglages finis sur un écran, il serait bon de pouvoir verrouiller cet écran en n'autorisant que sa lecture pour éviter toute manipulation accidentelle. L'utilisateur doit donc avoir la possibilité de choisir le mode d'ouverture de chaque écran en fonction des décisions prises par un responsable.

Ce mode de fonctionnement demande une gestion particulière des écrans GMS. Un **serveur gestionnaire d'écrans** a été créé à cet effet : **xvivetat**. Le rôle principal de *xvivetat* est de **gérer les droits d'ouverture et de fermeture d'un écran GMS**. Aussi, il doit être capable de donner, sur demande, les informations concernant la gestion des écrans.

8.1 L'aspect communication

Le gestionnaire peut communiquer avec les écrans soit par des procédures à distance (RPC) soit par des événements X. Ce serveur existait déjà dans la première version et le support de communication choisi à l'époque était les procédures à distance, mais ce choix présentait un défaut. Certains services demandaient l'utilisation de commandes systèmes qui ne pouvaient être exécutées que par le gestionnaire. La commande *kill* était, par exemple, utilisée pour "tuer" le processus associé à un écran. Mais lorsque le service devait être exécuté sur une machine distante, il pouvait créer des processus "fantômes" (*zombies*) sur la machine où se trouvée le gestionnaire, ce qui "polluait" le système. Certains services ne pouvaient donc pas être exécutés à distance.

Pour rester homogène avec le support de communication choisi dans la version actuelle, nous avons adopté la **communication par événements X**. La figure C.35 nous montre le principe de la communication gestionnaire - écrans GMS. Xvivetat est un client X qui reste connecté en permanence au serveur X local. Il communique avec les écrans GMS par envoi de messages intégrés dans le flux des événements X. Les écrans GMS peuvent se trouver sur la machine locale ou sur une machine distante. Ils envoient des requêtes (avec ou sans réponse) au gestionnaire d'écrans qui va les traiter localement, évitant ainsi les problèmes liés à l'exécution à distance des commandes systèmes.

L'espace réservé au transport de données dans un événement X est de 20 octets (§1.3.3.4) qui sont répartis dans une structure de données de type ***Ev_Xvivetat*** de la manière suivante :

- *8 octets pour le nom de la machine hôte (hostname) ;*
- *1 octet pour le numéro du serveur X (chapitre B, §1.3.3.1) associé à l'écran GMS ;*
- *2 octets pour le numéro de l'utilisateur ;*
- *4 octets pour le numéro de la fenêtre qui identifie le client X de l'écran GMS ;*
- *4 octets pour le nom de l'écran GMS ;*
- *1 octet pour le type de requête (ouverture écran, verrouillage écran, fermeture écran...).*

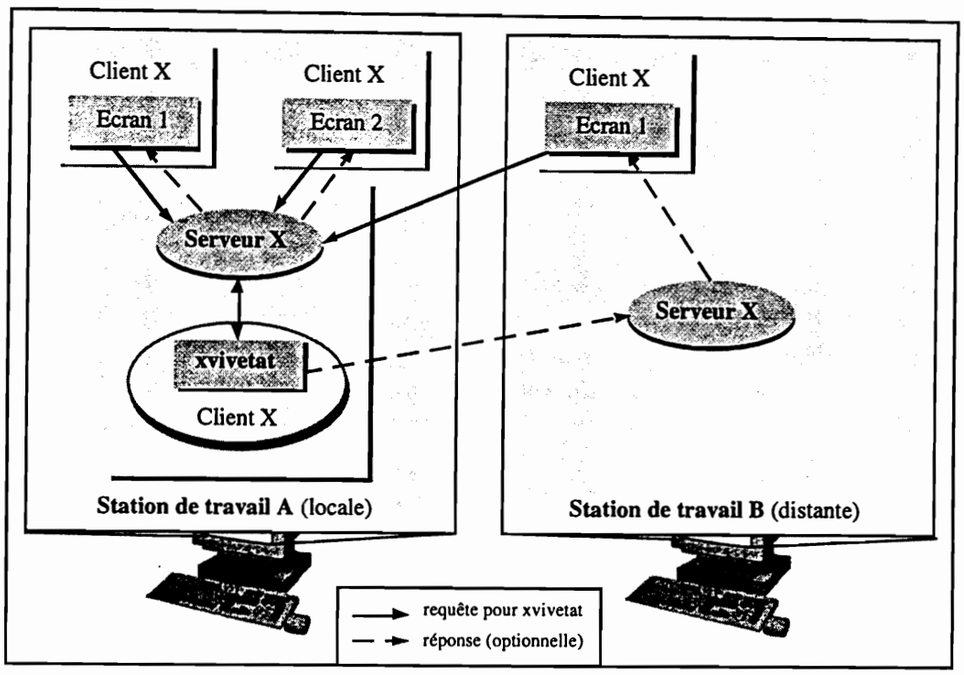


Figure C.35- Communication écrans GMS - gestionnaire d'écrans

8.2 La phase d'initialisation

Le serveur gestionnaire d'écrans est lancé une seule fois, à l'initialisation du système informatique, et son unicité doit être impérativement préservée. En effet, le lancement de plusieurs serveurs serait nuisible à la gestion des écrans GMS. Chaque station de travail possède un script de démarrage qui initialise une session X. Nous avons insérer la commande de lancement de xvivetat dans ce script. Chaque station peut donc lancer un gestionnaire d'écrans. Pour préserver son unicité nous avons réaliser un système de **verrouillage par fichier.**

Lors du premier lancement de *xvivetat*, celui-ci créera un fichier de verrouillage, "**.xvivetat.lock**", qui verrouillera tout les lancements postérieurs. Lorsqu'un deuxième serveur de gestionnaire d'écrans va être lancé, *xvivetat* vérifie auparavant qu'aucun fichier de verrouillage ne se trouve dans le répertoire de ressources ; si c'est le cas, le lancement du serveur est annulé. Le fichier de verrouillage (figure C.36) contient les références du serveur X associé à *xvivetat* (chapitre B, §1.3.3.1) et le numéro de la fenêtre (*window Id*) qui identifie son client X. Ces informations vont être utilisées par les clients X associés aux écrans GMS pour se connecter au serveur X associé au gestionnaire d'écrans.

Nous avons créer un fichier de ressources, "**.xvivetatrc**", où l'administrateur du système pourra écrire tous les droits conférés aux écrans GMS. Sa structure est représentée sur la figure C.36. Il contient le nom de tous les écrans GMS ; la liste des machines qui sont autorisées à ouvrir ces écrans, la liste des utilisateurs qui sont autorisés à travailler sur ces écrans et le nombre d'écrans qui peuvent être ouverts pour chaque mode d'ouverture (commande, mesure ou test). *Xvivetat* lira le fichier de ressources lors de son initialisation pour mémoriser les droits de chaque écran GMS.

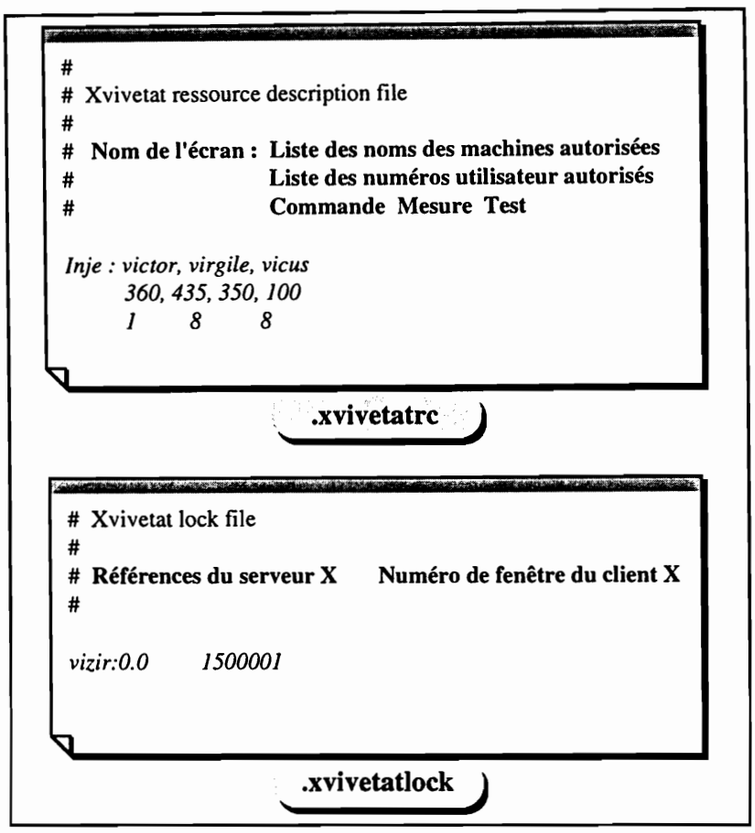


Figure C.36- Structure des fichiers de ressources et de verrouillage de xvivetat

Par ailleurs, lorsque l'utilisateur veut ouvrir un écran GMS, ce dernier envoie une requête à *xvivetat* (§6.3.1) qui vérifie les droits d'ouverture ; si les droits sont conformes, l'écran est ouvert et *xvivetat* l'insère dans sa liste d'écrans ouverts. Lorsque l'utilisateur ferme un écran GMS, ce dernier envoie une requête à xvivetat (§6.3.2) qui supprime l'écran de sa liste.

Le gestionnaire d'écrans dispose de plusieurs commandes utilisateurs qui permettent de "tuer" le gestionnaire d'écrans (*xvivetatkill*), de verrouiller ou déverrouiller un écran en commande (*xvivetatlock*, *xvivetatunlock*) et de connaître la liste de tous les écrans ouverts avec le nombre d'écrans pour chaque mode d'ouverture (*quiecran*).

9 Traitement des messages d'alarme

Dans notre premier système, chaque émetteur d'alarmes envoyait lui-même son message vers le média voulu (haut-parleur ou fichier). Cependant, si plusieurs émetteurs détectaient le même défaut, le même message était reproduit plusieurs fois sur les hauts parleurs ou le fichier. De plus, ce dispositif interdisait à un *concentrateur* de faire part d'une alarme n'ayant pas d'accès direct aux médias. C'est pourquoi, nous avons décider de réaliser **un ou plusieurs gestionnaires d'alarmes**, qui centralise(nt) tous les messages d'alarme pour les distribuer, après traitement, aux médias concernés.

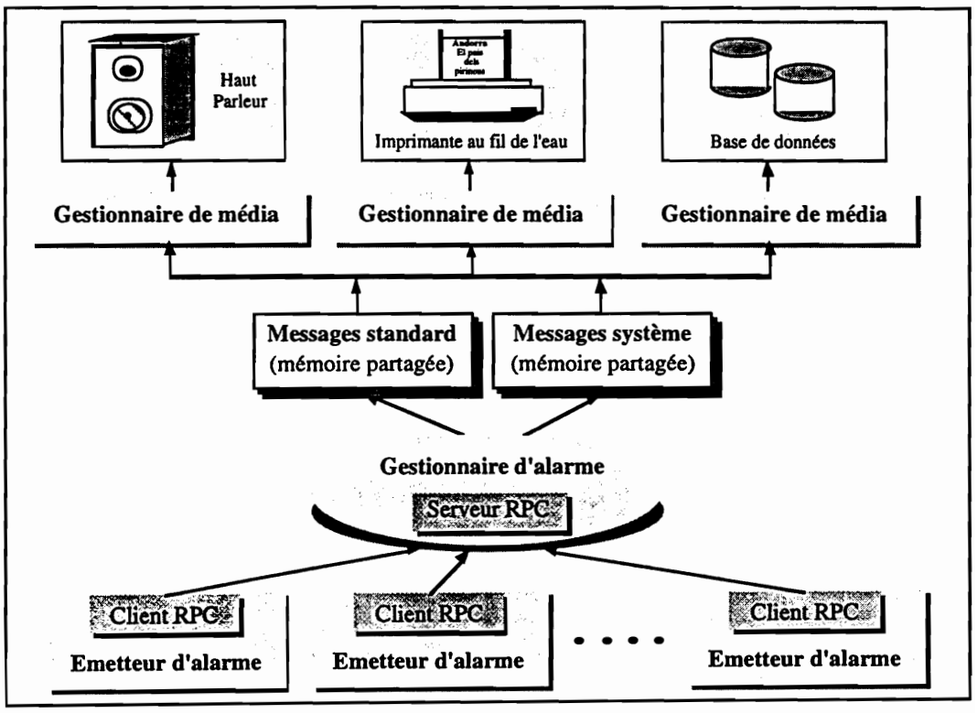


Figure C.37 - L'architecture de la gestion des alarmes

9.1 Les médias

Les messages sont dirigés vers trois médias différents correspondant à trois niveaux de réaction de l'opérateur :

- **Le haut-parleur** (H.P.)
  Il sera utilisé pour les messages nécessitant une réaction ou, au moins, l'attention immédiate des opérateurs. Par exemple, l'action sur une commande interdite provoque un "bip" sonore ne nécessitant aucun archivage mais informant l'opérateur que son action ne sera pas suivie d'effet.
- **L'imprimante au fil de l'eau**
  Elle sera utilisée pour les messages à consultation immédiate (le bruit de l'imprimante alerte l'opérateur). Par exemple, l'ouverture d'un écran est notée sur l'imprimante mais n'est pas suivie d'un message sonore ou d'un archivage dans la base de données.

- **La base de données**
  Les messages archivés auront l'avantage de pouvoir être facilement corrélés à des mesures ou à des commandes effectuées, mais l'accès à la base restera toujours plus complexe que la consultation d'un listing. La base de données sera plutôt perçue comme le média de l'historique du système (§10) plutôt que comme un média d'alarmes. Par exemple, la remontée d'une mesure en alarme est archivée uniquement dans la base.

Les messages auront la possibilité d'être dirigés simultanément vers plusieurs médias. L'arrêt d'un concentrateur, par exemple, sera signalé par tous les médias disponibles. Contrairement à la base de données, le haut-parleur et l'imprimante sont des médias à effet immédiat. Leur traitement sera donc prioritaire sur celui de la base.

Les messages à consommation immédiate (H.P. ou imprimante) seront émis suivant une périodicité raisonnable pour la consultation de l'opérateur (quelques secondes). De même, les messages répétitifs seront imprimés avec une périodicité de quelques secondes. Afin d'éviter une surcharge inutile de l'historique, la base de données ne stockera pas les messages identiques ayant pratiquement la même date d'émission. Le gestionnaire d'alarmes surveillera donc la fréquence d'émission des messages. Il pourra éventuellement préciser le nombre de fois qu'un message a été reçu depuis sa dernière émission.

9.2 Le gestionnaire d'alarmes

Nous venons de voir que le gestionnaire d'alarmes traite des alarmes en provenance de plusieurs sources et gère plusieurs médias. La gestion des alarmes peut se faire de deux façons :

- **Par gestion monomédia**
  Cette gestion offre une bonne modularité et permet une gestion optimum du média. Par contre, elle oblige les clients à choisir parmi plusieurs gestionnaires et, éventuellement, à dupliquer les envois des messages.
- **Par gestion multimédia**
  Cette gestion assure le synchronisme. Le client n'envoie qu'une seule fois le même message et ne choisit pas de gestionnaire. Tous les messages sont sélectionnés et traités par le même gestionnaire ce qui garantit l'unicité de l'algorithme et du traitement. Le seul inconvénient, mineur, est que cette gestion nécessite un algorithme assez lourd.

Le gestionnaire le plus adapté à notre application Vivitron est le gestionnaire **unique** et **multimédia**. Nous avons finalement un gestionnaire d'alarmes possédant les fonctionnalités suivantes :

- disposer d'un module unique de réception ;
- être capable d'identifier et de décoder un message ;
- disposer d'un sélecteur de média ;
- effectuer un traitement spécifique pour chaque média ;
- ne jamais être bloquant pour le reste du système.

Pour concilier ces fonctionnalités, nous proposons l'architecture modulaire de la figure C.37, composée par un module de réception qui alimente un module de gestion pour chaque média. Le rôle du gestionnaire sera de :

- dater le message si cela n'a pas été fait (message système) ;
- identifier les messages qui sont codés ;

- envoyer les messages vers les différents médias au plus vite et par ordre de priorité ;
- ne pas se bloquer si un média n'est pas disponible.

9.3 L'aspect communication

L'indisponibilité du serveur gestionnaire d'alarmes ne doit pas bloquer les clients. Il est préférable de rendre les clients muets pour éviter de voir des messages parasites envahir les écrans. La communication par **RPC** en mode **UDP**, entre les émetteurs d'alarmes et le gestionnaire, semble la plus appropriée. La communication entre le gestionnaire d'alarmes et les gestionnaires de média se fera par une **mémoire partagée**.

Les messages d'alarme peuvent provenir soit de l'application soit du système. Si le message provient de l'application, le contenu est connu d'avance sinon le contenu est à priori inconnu. Cette différence nous amènera à différencier le traitement des messages, ne serait-ce qu'à cause de l'impossibilité de transformer un message quelconque en un signal sonore.

9.3.1 Les messages standard

Ces messages sont connus et stockés dans une base incluant les données sonores. Ils sont numérotés pour pouvoir être identifiés par les applications qui les utilisent. Au niveau du gestionnaire d'alarmes, tous ces messages sont stockés dans une mémoire commune, sous forme d'un numéro accompagné d'une chaîne de caractères, d'un fichier audio et d'un état. Cet état est tenu à jour par le propre gestionnaire et indique si le message doit être pris en compte.

Les gestionnaires de média à **consommation immédiate** vérifient régulièrement s'il y a des messages à envoyer. Ce mode de travail en scrutation se justifie de part la simplicité qu'il apporte, du fait du temps nécessaire à la consommation d'un message. En effet, un message sonore, ou l'impression d'une chaîne de caractères, dure au minimum une seconde. En conséquence la fréquence de scrutation peut être de l'ordre de la seconde, ce qui ne perturbera en rien le système. Un système plus complexe, tel que des RPC ou des événements X n'apporterait rien d'autre que de la complexité.

Pour les gestionnaires de média à **consommation différée** (O_2) les messages devant être pris en charge proviennent autant du gestionnaire d'alarmes que du gestionnaire d'historiques (§10). Comme les gestionnaires de média ne travaillent que sur un flux unique, nous allons intégrer les messages d'alarme dans le flux général de données entrant dans la base O_2.

9.3.2 Les messages système

Le contenu de ces messages est à priori inconnu mais nous pouvons limiter sa longueur. Ils sont placés dans un tampon circulaire, non accessible par le gestionnaire du haut-parleur. Pour ne pas bloquer le système, le gestionnaire agira sans attendre la consommation des messages. Généralement, les messages les plus importants sont les derniers reçus ; c'est pourquoi, ils seront tous stockés dans une mémoire circulaire, même s'ils écrasent un message non encore diffusé. Ces messages seront numérotés et chaque gestionnaire de média parcourra le tampon à la recherche de tous les messages le concernant et veillera de ne pas émettre le même message plusieurs fois de suite.

10 L'historique

L'enregistrement de l'historique est concentré sur une seule machine, de manière à se débarrasser de problèmes d'accès concurrents à la base et surtout de faire l'économie d'une licence O_2.

L'historique du système englobe la totalité des informations archivées, en cours de session, dans la base de données O_2. Il enregistrera toutes les commandes, un échantillon de mesures suivant une périodicité de quelques minutes et divers messages d'alarme. Ces derniers peuvent être archivés à tout moment, mais les *paramètres* peuvent être archivés dans deux contextes différents :

- **En fonctionnement normal**
  Tous les *paramètres* sont systématiquement archivés, sauf les commandes qui ne sont pas effectuées par l'opérateur (commandes d'asservissement). Chaque mesure est échantillonnée, pour l'archivage, sur une période de temps qui lui est propre. Les commandes sont archivées dès quelles sont exécutées.
- **Sur demande**
  Toute **trace** d'un *paramètre* peut être archivée dans la base ou visualisée sur l'écran.

Le système d'historique sera conçu en fonction de cette diversité des sources, de grosses variations du débit en entrée (fort débit pour les mesures et faible débit pour les commandes), de la facilité d'accès aux archives et des contraintes propres à O_2 (problèmes d'objets temporaires, de transactions...).

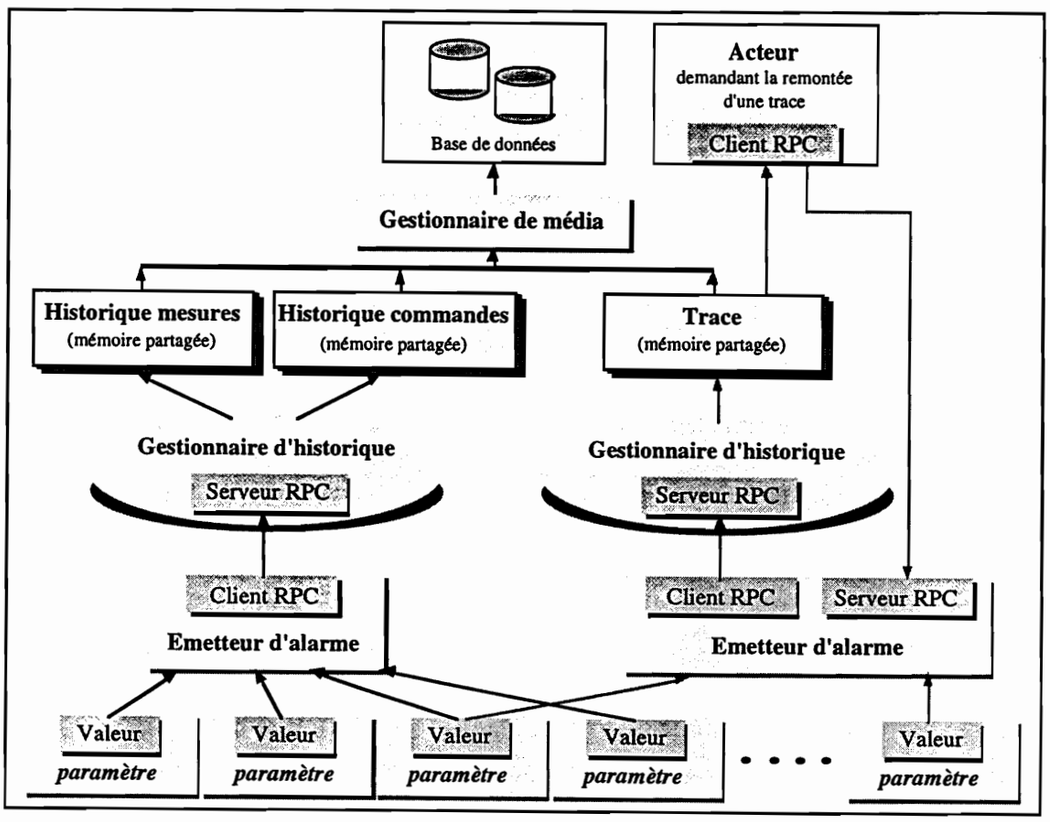


Figure C.38 - L'architecture du système d'historique

10.1 Le système d'historique

L'historique doit archiver des informations dans deux contextes différents qui demandent des traitements différents. Il y aura donc un serveur d'archivage pour chaque contexte :

- **Le serveur d'historiques**
  Il récupère les valeurs des *paramètres* envoyées périodiquement par les clients d'historique des *concentrateurs*. Ces valeurs sont ensuite enregistrées dans la base de données.
- **Le serveur de traces**
  Il récupère les valeurs des *paramètres* demandées ponctuellement par un acteur externe. Ces **traces** doivent être disponibles pour cet acteur.
- **Le gestionnaire d'alarmes**
  C'est lui qui gère les messages d'alarme (§1.2.4). Les messages d'alarme seront consommés en donnant au gestionnaire de média de la base de données l'accès aux deux tampons du gestionnaire d'alarmes (messages standard et messages systèmes).

10.2 L'aspect communication

10.2.1 Communication acteur - serveur

Tous les acteurs devant émettre des messages vers les serveurs d'historiques utilisent le même protocole de communication. Pour des raison d'homogénéité (§9.3) et de simplicité, nous utiliserons les **RPC** en mode **UDP**. A priori, nous pourrions utiliser les *formulaires* (§6.3) qui permettent de transférer un bloc de *paramètres* en un seul appel. Mais cette méthode entraîne deux inconvénients majeurs :

- Nous serions obligés de gérer un tampon au niveau des *concentrateurs* en attendant que le *formulaire* soit plein. En cas de défaillance, nous perdrions les derniers *paramètres* à archiver, sans doute les plus importants.
- La remontée d'une *trace* se ferait valeur par valeur, ce qui obligerait à un traitement spécifique pour chacune d'entre elle par le serveur.

Nous sommes donc amenés à définir un autre format de transfert de données entre les serveurs d'historiques et les acteurs. La règle fixée est que chaque *paramètre* remonté est véhiculé par un seul RPC, mais que ce dernier peut contenir plusieurs valeurs.

10.2.2 Communication serveur - O2

Le serveur d'historiques reçoit un flot variable d'informations à enregistrer. Il ne doit jamais bloquer un client, ni même le retarder. La consommation des messages par O_2 doit être immédiate. Si l'enregistrement dans O_2 "patine", il faut alors accepter de perdre des informations pour ne pas perturber le fonctionnement du système au risque de perdre des informations importantes. Ce rôle de

fusible est dévolu à une mémoire tampon créée et initialisée au démarrage de la machine. Son fonctionnement est similaire aux tampons circulaires du gestionnaire d'alarmes (§9) : tout nouveau message doit pouvoir être écrit dans le tampon et dans les meilleurs délais, même s'il faut écraser un message plus ancien et pas encore consommé. Nous aurons un tampon différent pour les traces, les mesures et les commandes (figure C.38).

Les messages d'historique sont stockés sous forme d'une structure de données contenant :

- *le symbole du paramètre ;*
- *la valeur et la date ;*
- *un état indiquant si le message a été consommé ou non par O_2 ;*
- *un champ protégeant le tampon contre les accès concurrents.*

Les algorithmes doivent être écrits de façon à minimiser le temps d'occupation du tampon. Pour cela, le gestionnaire de média de O_2 acceptera de perdre du temps pour recopier quelques messages afin de ne pas lier l'occupation du tampon au temps de réponse de O_2.

Par ailleurs, afin de gagner du temps sur l'écriture dans O_2, il est souhaitable de traiter plusieurs messages en une seule requête. Le gestionnaire de média disposera donc d'un tampon local de même structure que la mémoire partagée.

11 Le démarrage du système à partir du SGBDOO O2

Nous avons voulu intégrer dans la base, en plus de la **description** du système de contrôle et commande, le **code** des programmes devant gérer ce système. Pour éviter une fastidieuse réécriture du même code pour chaque nouveau paramètre, nous avons réalisé un outil qui permet au SGBDOO O_2 de générer ce code.

Dans ce chapitre nous insisterons sur l'aspect **génération de code par la base de données O_2**. Dans un premier temps, nous définirons les modules indispensables au démarrage du système et nous verrons comment ces modules sont modélisés dans la base. Ceux-ci doivent être compilés dans un environnement particulier pour obtenir les modules relogeables prêts à être chargés dans les *concentrateurs*. Cet environnement sera aussi intégré dans la base. Nous avons créé deux interfaces, l'une concernant O_2 et les *concentrateurs*, l'autre concernant O_2 et les applications GMS. Nous développerons surtout les caractéristiques de la première interface car c'est le travail qui m'a été confié, et nous aborderons brièvement la deuxième interface pour une meilleure compréhension du rôle du SGBDOO O_2.

11.1 L'interface O2 - concentrateurs

Cette interface doit garantir une cohérence entre tous les modules du système de contrôle et commande. Pour cela, le SGBDOO O_2 doit être doté d'un mécanisme de génération de modules dans lequel les modifications apportées aux données stockées dans la base ne soient prises en compte qu'après la régénération de tous les modules affectés. Le principal choix est de stocker et d'éditer les fichiers sources directement dans la base, ou de les éditer sous UNIX à partir d'un squelette fourni par O_2. L'existence des *paramètres* est ainsi vérifiée par la base, ce qui élimine l'introduction d'une éventuelle erreur de paramétrage

Comme nous avons décidé de gérer les fichier sources dans O_2, il est naturel de contrôler la compilation à partir de O_2. Ce n'est pas une obligation, mais c'est plus confortable. Les fichiers sources produisent, après compilation, des **modules objet** qui sont stockés dans des répertoires spécifiques. Le stockage de ces modules à l'extérieur de la base de données permet un fonctionnement, même dégradé, de chaque module en l'absence de O_2. Ces modules sont chargés dans les *concentrateurs* lors de leur démarrage. Ils se divisent en deux catégories :

- les **modules standards**, communs à tous les *concentrateurs* et donc indépendants des données stockées dans la base (noyau VxWorks, clients RPC, fonctions d'acquisition, ...);
- les **modules spécifiques**, propres à chaque *concentrateur* et donc dépendants des données définies dans la base (fonctions de mesure et de commande, fonctions d'alarme et d'asservissement, fonctions d'initialisation des *paramètres*, ...).

Avec cette nouvelle utilisation du SGBDOO O_2, les fichiers de description des paramètres (les fichiers *o2resume*) et leur tâche d'interprétation (*conclnit*) (§1.3.2) ne sont plus nécessaires pour le démarrage des *concentrateurs*. Chaque *concentrateur* doit aller chercher ses propres paramètres de démarrage dans un fichier spécifique généralement appelé **startup**. Ce fichier peut être propre à chaque

concentrateur ou commun à tous les *concentrateurs*. Dans notre nouveau système nous avons décidé de créer un premier fichier commun à tous les *concentrateurs*, **startup.cmd**, pour charger les *modules standards* et un deuxième fichier spécifique à chaque concentrateur, **start.v2**, pour charger les *modules spécifiques*. Les *concentrateurs* appellent le fichier *startup.cmd* qui chargera leurs modules objet et lancera le fichier *start.v2* correspondant au *concentrateur* en cours d'initialisation. Notons que le chemin d'accès du fichier *startup.cmd* est communiqué au *concentrateur* lors de sa phase de configuration, tandis que les chemins d'accès des fichiers *start.v2* sont connus de tous par une variable globale qui est modifiée constamment par le *concentrateur* en cours d'initialisation.

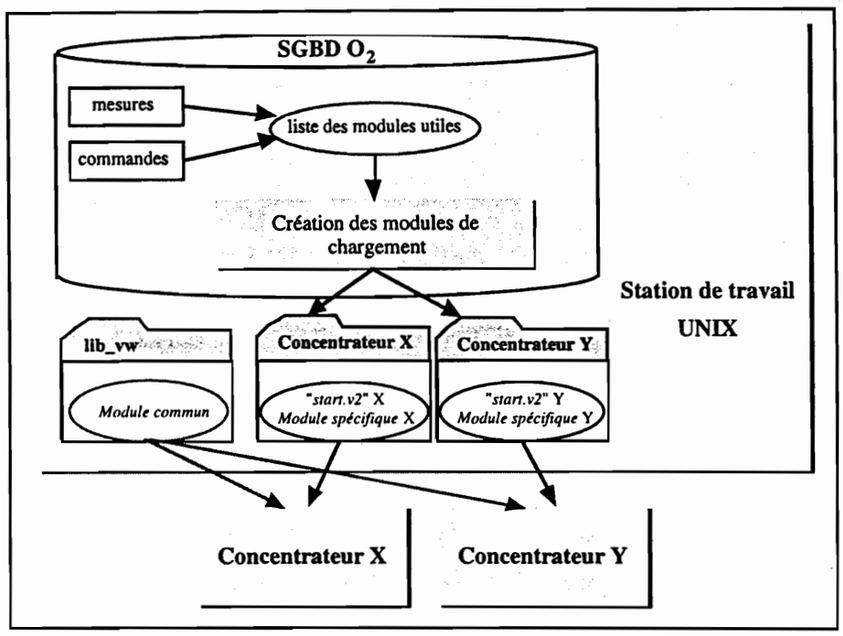


Figure C.39 - Initialisation des concentrateurs

Le nouveau système ainsi créé se présente sous la forme du schéma de la figure C.39. Le SGBDOO O_2 génère le fichier *start.v2* ainsi que les *modules spécifiques*. Ces derniers sont construits à partir des mesures et des commandes stockées dans la base et sont placés dans des répertoires propres à chaque concentrateur. Les *modules standards*, quant à eux, sont déjà stockés dans un répertoire commun à tous les *concentrateurs*, appelé *lib_vw*. Pendant la phase de démarrage, chaque *concentrateur* va chercher ses modules objets et les *modules standards*.

La plupart des fonctions s'exécutant sur les *concentrateurs* font partie des *modules standards* et sont générées par le développeur. Seules quelques fonctions, dépendant des équipements gérés par chaque concentrateur, appartiennent aux *modules spécifiques*. Un tri doit donc être effectué parmi toutes ces fonctions pour repérer celles qui nous intéressent : les *fonctions spécifiques*. Celles-ci sont générées automatiquement par le SGBDOO O_2 dans la mesure du possible, sinon le codage est assisté.

Les paragraphes qui vont suivre développent chaque fonction spécifique en insistant sur l'aspect génération de code ; mais auparavant, nous allons voir comment est modélisé ce mécanisme de génération de code dans la base de données.

11.2 Génération du code concentrateur

Les programmes du *concentrateur* sont écrits en langage C ANSI et compilés sous un environnement VxWorks qui diffère légèrement de l'environnement de compilation UNIX. Il est important de connaître les principales étapes de la création d'un programme en langage C sous un environnement VxWorks pour bien modéliser ce mécanisme dans la base O_2.

11.2.1 L'édition du programme

Un programme écrit en langage C est une succession de déclarations et de définitions de fonctions, variables et types. Des commentaires et des directives d'inclusion pour le processeur peuvent parsemer le programme.

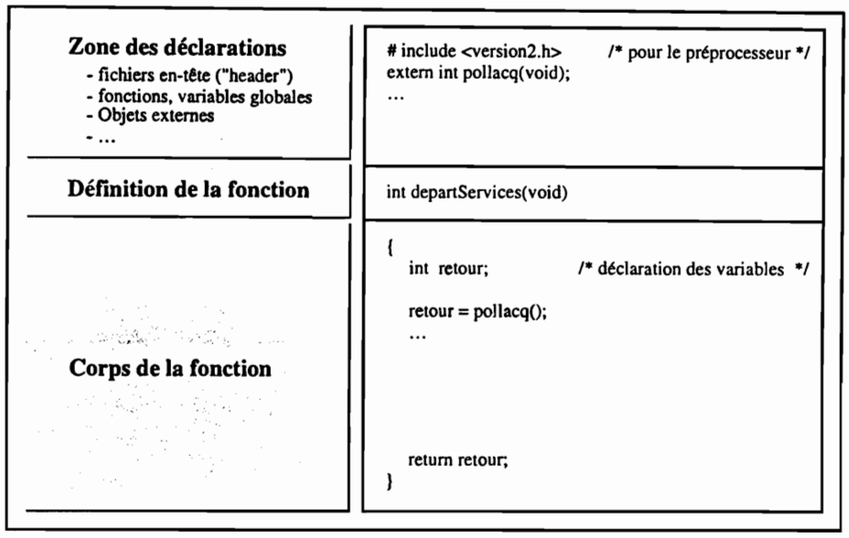


Figure C.40 - Structure générale d'un programme en langage C

La figure C.40 nous montre la "structure générale" d'un tel programme. Nous pouvons observer qu'il a une structure en trois parties : une zone de déclarations, la définition de la fonction principale et le corps de cette fonction. Nous noterons que le nom du programme principal n'est pas ***main***, comme le précise la norme du langage C, mais un nom choisi par le développeur. Cette caractéristique est propre à l'environnement VxWorks et nous en profiterons pour donner au fichier source le même nom que le programme principal.

11.2.2 La compilation

Une fois le programme source rédigé, il est traité par le préprocesseur qui, suivant les directives qui y sont contenues, substitue des symboles, inclut des fichiers sources, etc. Le compilateur prend la suite des opérations pour examiner chaque ligne, vérifier la syntaxe et générer un module objet. La compilation s'effectue sous VxWorks à l'aide de la ligne de commande suivante (la

commande ***cc68k*** est remplacée par la commande ***cc*** sous UNIX) :

cc68k *-o module_objet.o -c fichier_source.c* $(FLAGS) $(INCLUDE)

$(FLAGS) --> options (*-ansi -pedantic -Wall -fno-builtin -m68040 -m68040-only -DCPU=MC68040*) ;

$(INCLUDE) --> chemins d'accès des fichiers à inclure.

11.2.3 L'édition de liens

Pour produire le code exécutable du programme il faut rassembler les modules objet intermédiaires (dans le cas ou le programme est découpé en plusieurs fichiers sources) et les lier à diverses bibliothèques, dont la bibliothèque standard. C'est le rôle de l'**éditeur de liens** que l'utilisateur peut solliciter par la commande **ld** (environnement UNIX), ou **ld68k** (environnement VxWorks). Si le nombre de modules objets produits est trop important, l'éditeur de liens est aussi utilisé avec l'option ***-r*** pour les rassembler sur un seul module objet. La ligne de commande sous VxWorks est la suivante :

ld68k *-o librairie.o -r module1.o module2.o module3.o*

Sous UNIX, la commande ***cc*** ne correspond pas seulement à l'exécution du compilateur C. Par défaut, elle enchaîne automatiquement les différentes phases nécessaires à la production du fichier exécutable.

11.2.4 Représentation dans la base de données

L génération de chaque fonction du concentrateur est modélisée par un objet ***Fonction_Conc*** (§7.6). Il construit localement la structure générale de la fonction (figure C.40), excepté son corps qui est généré par une méthode de la classe ***Concentrateur***, et la stocke dans le champ *source* (au format texte) puis dans le champ *binaire* (au format binaire), après compilation. Si la compilation s'est bien passée, une méthode de la classe ***Concentrateur*** place le fichier source et le fichier objet correspondants à la fonction dans des répertoires particuliers. Ceci nous garantit le bon fonctionnement des *concentrateurs* en l'absence de la base de données.

Le développement d'une importante application comme le projet Vivitron se matérialise par une multiplicité de petits modules fonctionnels compilés séparément. Le problème qui se pose lors de la maintenance de l'application est de savoir quels sont les fichiers sources qui doivent être re-compilés lors d'une modification de l'un d'entre eux. La solution est d'établir le graphe de dépendances des fichiers sources de l'application. Dans le monde UNIX ou VxWorks ces dépendances sont gérées par la commande **make** et la description de ces dépendances se retrouve dans un fichier appelé **Makefile** et écrit par l'utilisateur. La base de données O_2 gère les dépendances entre tous les objets de la classe ***Fonction_Conc*** ainsi que les versions de ses instances. Un fichier *Makefile* sera tout de même reproduit par la base pour faciliter la maintenance des programmes en l'absence de celle-ci.

Les commandes exécutées sous UNIX sont stockées dans des instances de la classe ***Ligne_Commande*** sous forme de texte (classe ***Text***). Suivant l'environnement de compilation dans lequel nous nous plaçons, UNIX ou VxWorks, les lignes de commande changent. Ces deux environnements sont rangés dans deux objets nommés : ***std_env_unix*** et ***std_env_vw***, et sont définis par la classe ***Environnement_Pgm*** qui rajoute aux commandes leur environnement.

11.3 Les fonctions codées manuellement

Après avoir vu comment sont générées les fonctions des *concentrateurs*, nous allons découvrir de quelles fonctions il s'agit. Nous avons deux sortes de fonctions : les fonctions **générées automatiquement** et les fonctions **éditables par le programmeur**. Nous ne traiterons ici que ces dernières. Elles concernent les fonctions associées aux mesures et aux commandes, les fonctions d'asservissement, les fonctions d'alarme, les fonctions de sécurité et les fonctions associées aux horloges auxiliaires. Chacune a son propre algorithme qui varie suivant les équipements attachés à chaque concentrateur. Dans le meilleur des cas, la base ne peut proposer qu'une fonction par défaut sinon, elle autorise le développement manuel des fonctions tout en contrôlant la cohérence entre les *paramètres* de la base et ceux rentrés par le développeur. Ainsi, la fonction générée ne pourra accéder qu'à des *paramètres* existant dans le concentrateur.

11.3.1 L'accès aux paramètres et aux ordres : les alias

Chaque *paramètre* a une liste d'*ordres* qui lui est propre et par laquelle il peut accéder aux canaux physiques. Cependant, il est important d'éviter qu'un *paramètre* accède à des *ordres* qui ne lui appartiennent pas, sous peine d'accéder à des canaux inexistants ou de créer des conflits avec d'autres *paramètres*. Pour contrôler ce genre d'accès, nous avons introduit la notion d'**alias**. Un *alias* sert à référencer l'accès à un *paramètre* ou à un *ordre* (lu ou écrit) par une syntaxe et des symboles bien précis. L'accès devient alors transparent pour le programmeur qui n'est plus obligé de connaître le numéro des canaux, le nom de leurs fonctions d'accès (fonction d'acquisition, fonction de commande, etc.) ou le nom des champs d'accès liés aux structures de données des *paramètres*. Certaines règles doivent être respectées lors de l'accès aux *paramètres* et aux *ordres* :

- Une fonction *concentrateur* peut accéder à tous les *paramètres* du concentrateur.
- Un *paramètre* peut accéder à tous les *paramètres* de son *équipement atomique* et à tous ses *ordres*.
- Un *ordre* qui n'est pas connecté à un *paramètre* est inaccessible. L'*ordre* ne peut être utilisé que s'il est supervisé par un *paramètre*.

La syntaxe générale d'un *alias* est formé par le nom symbolique de la mesure, la commande ou l'*ordre* (tableau T.9) entouré par des arrobas ("@@"). Si l'alias fait référence à une fonction qui admet un argument en entrée, non connu de la base, il est suivi de cet argument mis entre parenthèses. L'utilisation des arrobas sert à référencer la partie de code à pré-traiter.

	nom symbolique	**exemple**
Mesure	[référence équipement]_[référence mesure]_M	Aimant_Courant_M
***Ordre* lu**	[référence *ordre* lu]_MO	Courant_MO
Commande	[référence équipement]_[référence commande]_C	Aimant_CourantDAC_C
***Ordre* écrit**	[référence *ordre* écrit]_CO	CourantDAC_CO

Tableau T.9 - La syntaxe générale d'in alias

Une phase de pré-traitement décode l'alias pour le remplacer par un appel adéquat dans le texte source. Il y a deux catégories d'appels : ceux concernant l'**accès au matériel**, par le biais des *ordres,* et ceux concernant l'**accès aux fonctionnalités** par le biais des *paramètres*. Mais, pour mieux comprendre la notion d'*alias*, nous allons prendre un exemple pour chaque type d'accès, en donnant l'*alias* et son appel correspondant.

L'accès aux *ordres* :

- Les fonctions qui effectuent l'acquisition d'une donnée sur un seul canal ;
  @@Acq_Courant_MO@@
  acqVadcCanal(adresse_lien, numéro_canal)
- Les fonctions qui exécutent une consigne sur un seul canal ;
  @@OnOff_CO@@
  cmdVmod(adresse_lien, numéro_canal, consigne)
- La lecture de la valeur d'un canal dans la table d'accès aux canaux (§3.4).
  @@Courant_MO@@
  accesVadc_0.canal[numéro_canal]

L'accès aux *paramètres* :

- Les fonctions qui envoient une commande sur un concentrateur ;
  @@Alim_CourantDAC_C@@(consigne)
  CommandeEnvoie(concentrateur, ***consigne****, adresse_commande)*
- La récupération de la consigne d'une commande ;
  @@Alim_CourantDAC_C@@
  Alim_CourantDAC_C.valeur_fraiche.physique
- Les fonctions qui stockent dans l'historique plusieurs valeurs d'une même mesure ;
  @@Alim_CourantDAC_M@@(nombre_mesures)
  msgHistoTraceMesures(adresse_mesure, ***nombre_mesures****, concentrateur)*
- Les fonctions qui stockent dans l'historique une seule valeur d'une même mesure.
  @@Alim_CourantDAC_M@@
  msgHistoTraceMesure(adresse_mesure, concentrateur)

11.3.2 La gestion des alias dans la base

Les *alias* sont modélisés dans la base par des objets de type ***Fct_Alias*** composés de trois attributs :

- *symbole* : qui récupère le nom symbolique du paramètre ;
- *ref_réelle* : qui contient la chaîne de remplacement réelle ;
- *ref_test* : qui contient une chaîne de remplacement pour faire des tests.

Les chaînes de substitution prévoient une place, repérée par une étoile, pour les alias contenant des arguments. Donnons un exemple :

- *symbole* = "Alim_CourantDAC_C";
- *ref_réelle* = "CommandeEnvoie(C_90, *, Vdac_0)";
- *ref_test* = "CommandeTest(C_90, *, Vdac_0)".

Les *alias* concernant les mesures et les commandes sont regroupés sous forme de liste d'objets dans la classe ***Equipement_Atomique***. Ils sont générés par la méthode **alias** de la classe ***Commande*** ou ***Mesure***. Ceux concernant les *ordres lus* ou les *ordres écrits* sont regroupés sous forme de liste d'objets dans la classe ***Ordre***. Ils sont générés par la méthode "*alias*" de la classe ***Ordre***.
Les *alias* sont utilisés par toutes les fonctions codées manuellement dans la base (fonctions associées aux mesures et aux commandes, fonctions d'asservissement, ...). Ces fonctions sont des objets appartenant à la classe ***Concentrateur***.

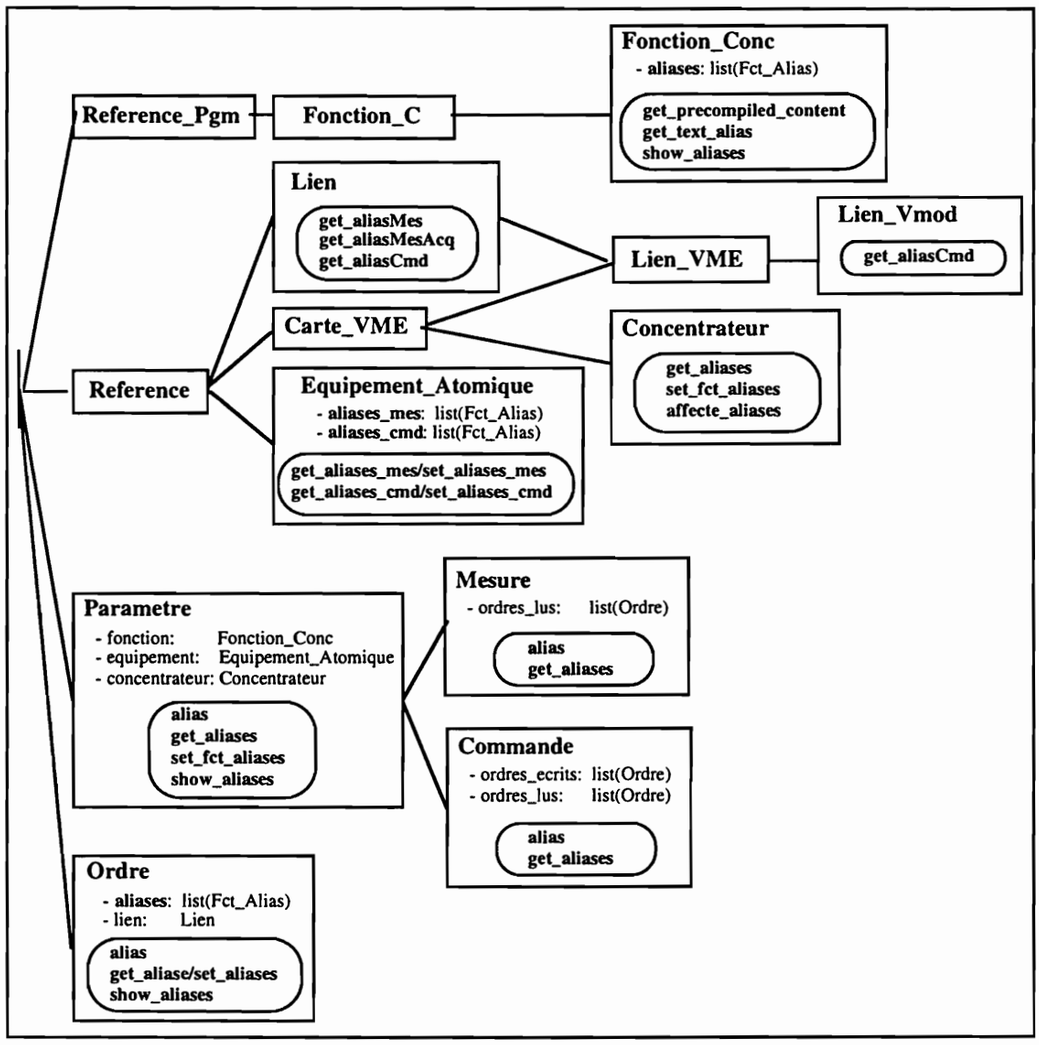


Figure C.41 - Gestion des alias dans la base

11.3.3 Les fonctions associées aux mesures

Rappelons que ces fonctions ont pour rôle de créer les mesures à partir d'*ordres lus* sur un même *concentrateur* et/ou de *paramètres* lus sur n'importe quel *concentrateur* (§3.3). Une mesure appartient à un équipement atomique mais reste accessible par le concentrateur. Elle peut accéder à tous les canaux physiques du système, en revanche, elle n'est associée qu'à une seule fonction.

Les fonctions associées aux mesures sont créées, éditées et mises au point au sein des objets de la classe ***Mesure***. Elles accèdent à tous les *ordres lus* du système via l'attribut *ordres_lus* (figure C.42a). Pour les mesures qui sont reliées à un seul *ordre*, et c'est le cas de la majorité des mesures, une fonction par défaut leur est affectée par la méthode *code_fctParDefaut*. Cette fonction est générée automatiquement mais peut être modifiée manuellement pour l'adapter à une mesure particulière. D'autres mesures, par exemple le **stripper feuille** (§3.3), sont composées de plusieurs canaux et bénéficient d'un traitement particulier, mais elles sont présentes dans de nombreux *concentrateurs*. Pour ces mesures nous avons créé des méthodes adaptées, *code_FctStripper* par exemple, qui évitent de réécrire manuellement le code chaque fois qu'elles sont utilisées.

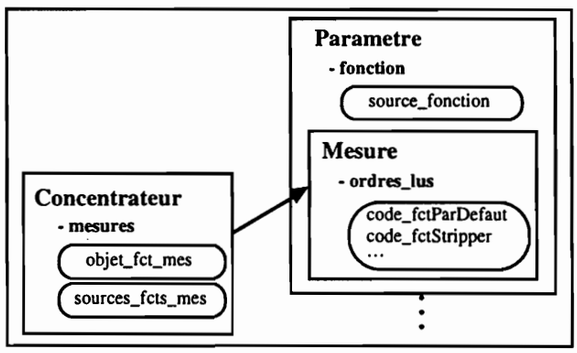


Figure C.42a - Les fonctions associées aux mesures

11.3.4 Les fonctions associées aux commandes

Les fonctions associées aux commandes se construisent pratiquement de la même manière que les fonctions associées aux mesures. La seule différence est que les fonctions de commande créent les commandes à partir d'*ordres écrits* et d'*ordres lus*.

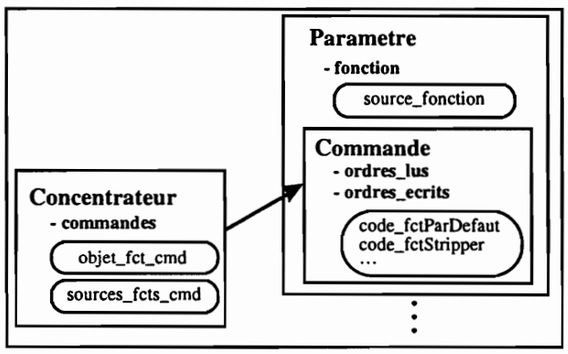


Figure C.42b - Les fonctions associées aux commandes

11.3.5 Autres fonctions

Les fonctions d'asservissement, les fonctions d'alarme, les fonctions de sécurité et les fonctions de l'horloge auxiliaire. Toutes ces fonctions doivent avoir accès à tous les *paramètres* gérés

par le concentrateur. C'est donc au sein de la classe ***Concentrateur*** qu'elles seront éditées et mises au point. Chaque fonction est un attribut de la classe ***Concentrateur***.

11.4 Les fonctions codées automatiquement

Il nous reste à voir les fonctions qui sont générées automatiquement par la base, c'est-à-dire les fonctions qui n'ont recours à aucune assistance manuelle afin de garantir un comportement homogène d'un *concentrateur* à l'autre. L'algorithme de ces fonctions est le même pour tous les *concentrateurs*, sauf son contenu qui change en fonction des équipements du concentrateur. Ces fonctions concernent la tâche d'acquisition *tPollAcq*, les fonctions d'initialisation des *concentrateurs* et le dispositif de démarrage.

11.4.1 La tâche d'acquisition tPollAcq

La tâche *tPollAcq* est propre à chaque *concentrateur* car elle fait appel aux fonctions d'acquisition de tous les *liens* reliés au concentrateur. Nous allons l'intégrer dans la base en tant qu'attribut de la classe ***Concentrateur*** (*pollacq_fct*). Son code est généré par la méthode *code_pollacq* qui va scruter tous les *liens VME* et tous les *liens Série* attachés au *concentrateur* en négligeant le lien VME de type ***Vdac*** qui n'a pas de fonction d'acquisition car ce n'est pas une carte d'acquisition. Pour chaque *lien*, elle va déclencher la méthode *code_pollacq* de la classe ***Lien*** dont héritent les classes ***Lien_Serie*** et ***Lien_VME*** (figure C.43). Cette méthode recherche le type de carte (Vadc, Vmod, Incaa, ...) et génère le code de l'appel à la fonction d'acquisition.

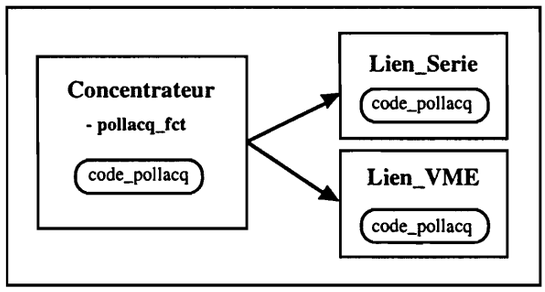


Figure C.43 - La tâche tPollAcq

11.4.2 Les fonctions d'initialisation

Les mesures, les commandes, les *liens*, les canaux..., sont des variables globales du *concentrateur* qu'il faut créer puis initialiser. Ce rôle est tenu par des fonctions d'initialisation qui concernent :

- l'auto-configuration des *liens* (*initAutoConf()*);
- l'horloge auxiliaire (*initAuxClock()*);

- l'accès aux canaux physiques (*initConcCanaux()*);
- l'initialisation des *liens* (*initConcLiens()*);
- la description des *paramètres* (*initConcParametres()*);
- l'initialisation des variables globales (*initconcVariablesGb()*);
- la fonction qui lance toutes les fonctions d'initialisation (*initConcentrateur()*).

Toutes ces fonctions sont des objets gérés par la classe ***Concentrateur*** sous forme d'une liste d'objets, (liste accessible par l'attribut *init_conc_fcts*) sauf les deux premières qui sont gérées comme deux objets accessibles respectivement par les attributs *init_auto_conf_fct* et *init_aux_clock_fct*.

11.4.3 Le départ des tâches de service

Le démarrage du *concentrateur* fait appel à un ensemble de scripts et de fonctions assurant respectivement le chargement des différents modules dans les *concentrateurs* et le lancement des tâches de service (serveur de formulaires, serveur d'alarmes, *tPollAcq*, ...).

Les scripts sont composés des fichiers *startup.com* et *start.v2* vus précédemment. Nous rappellerons donc que c'est *start.v2* qui est pris en charge par O_2. Son rôle est de :

- détecter les liens qui sont présents dans le *concentrateur* et créer leur symbole correspondant en cas de présence ;
- charger les modules objet générés par O_2 ;
- lancer toutes les fonctions d'initialisation ;
- lancer les différents services ;
- démarrer l'horloge auxiliaire.

Les fonctions qui lancent les services sont prises en charge par l'attribut *depart_fcts* de la classe ***Concentrateur***.

11.5 La génération d'une application utilisateur

Pour finir, nous montrerons la démarche à suivre pour générer une application utilisateur avec le générateur de code de la base de données O_2. Cette démarche (figure C.44) se divise en trois étapes :

- **Création de la base des équipements matériels**
  créer les *concentrateurs* (1) ;
  créer les *liens* (2) ;
  valider et nommer les *ordres* (3).
- **Création de la base des équipements matériels**
  créer les *équipements atomiques* (4) ;
  créer les *paramètres* (5) ;
  connecter les paramètres à la base des équipements (6) ;
  éditer les fonctions des *paramètres* ;
  générer le code et compiler.

- **Création des écrans GMS**
  dessiner les écrans sous *Draw* (chapitre B, §3.4.1) ;
  créer les *équipements composites* (7) ;
  créer l'application GMS (8) ;
  connecter l'application à la base des équipements (9) ;
  générer le code GMS.

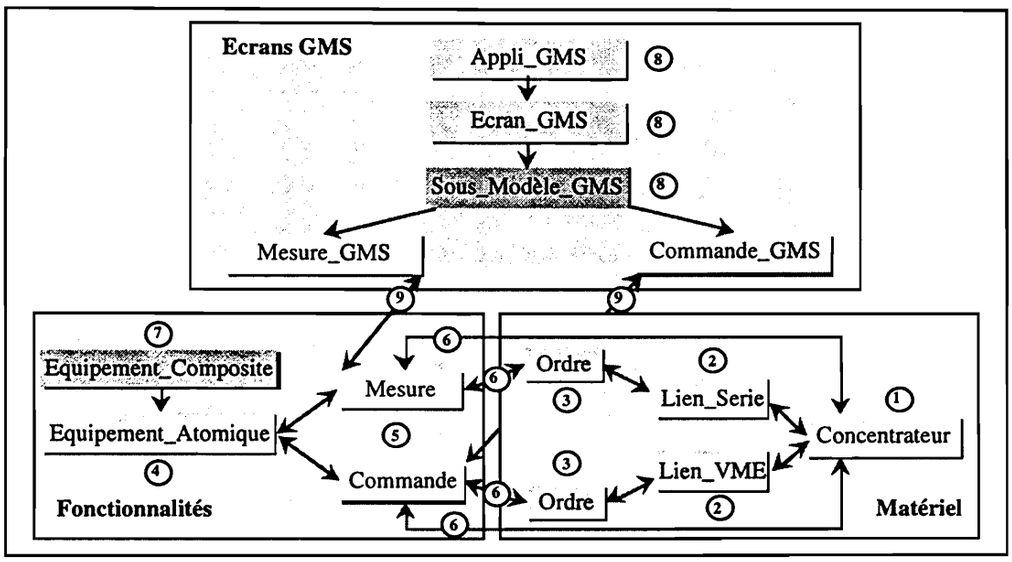


Figure C.44 - Les différentes étapes de la création d'une application utilisateur

12 Apports du nouveau système

Le premier système de contrôle et commande a été réalisé parallèlement à la mise en place et aux tests des équipements. Nous devions alors faire nos tests et nos mises au point en même temps que les travaux de construction de l'accélérateur avançaient. L'indisponibilité et la complexité de la machine nous interdisaient de faire des tests complets. C'est pourquoi nous ne pouvions pas garantir une totale fiabilité sur notre système (§2.4).

D'un autre côté, au fur et à mesure que les travaux de construction arrivaient à leur fin, de nouveaux services nous étaient demandés pour contrôler les nouveaux éléments de la machine. L'intégration de ces services dans notre système a entraîné plusieurs remaniements.

Les très courts délais qui nous étaient impartis nous ont obligés à produire des services immédiatement opérationnels. Cette situation ne nous laissait que peu d'opportunités pour pousser notre travail d'analyse et fédérer les compétences de tous.

Au cours de l'exploitation de ce premier système, plusieurs défauts sont apparus :

- La remontée systématique et périodique des mesures occasionnait des saturations sur les affichages graphiques des stations de travail (§1.3). Ces blocages pouvaient perturber grandement l'envoi des commandes.
- Le *concentrateur* ne pouvait pas signaler son état de fonctionnement aux écrans graphiques qui lui étaient associés. L'opérateur ne pouvait donc voir l'état réel du système et, en cas de panne, il ne pouvait pas réagir instantanément.
- GMS impose une période d'échantillonnage des mesures trop lente (chapitre B, §3.4.31), ce qui nous donnait une mauvaise perception de l'évolution temporelle des mesures.
- La dispersion des différents modules entraînait, lors d'une modification, un risque réel d'introduire un défaut car il devenait très difficile de garantir la cohérence entre les modules.

L'expérience acquise lors de ce premier travail nous a permis de mieux cerner le contrôle et la commande de la machine. Cette précieuse expérience nous a poussé à revoir l'architecture informatique globale du système afin de réaliser un nouveau système pour **corriger les défauts du premier, ajouter des services, améliorer la fiabilité et faciliter la maintenance.**

12.1 Nouveaux services

Nous avons rajouté une couche logicielle aux mesures et aux commandes pour supprimer la lourde gestion des *mesures calculées* (§2.6.1) et des *commandes calculées* (§2.6.2). Le traitement de ces données est grandement simplifié et peut être intégré au plus près des capteurs et actionneurs : dans les *concentrateurs.* Au sein de la couche de mesure ou de commande nous avons fait naître les **fonctions associées aux mesures ou aux commandes**. Ces fonctions peuvent accéder à plusieurs canaux, ce qui découple le canal physique de la mesure ou de la commande. Cet avantage nous permet de donner une description des *paramètres* plus abstraite permettant de ne véhiculer entre les *concentrateurs* et les écrans que des données pertinentes.

Un **gestionnaire d'écrans**, xvivetat (§8), gère les droits conférés aux écrans GMS : exécution des commandes, lecture des mesures ou mode test. Il permet notamment d'autoriser l'ouverture en

commande d'un écran sur un poste spécifique et d'interdire les commandes sur les autres postes. Le poste de contrôle de l'injecteur ne pourra commander que les écrans de l'injecteur, par exemple.

Les **formulaires** (§6.3) autorisent les reprises sur panne du système ainsi que le couplage étroit entre écrans jumeaux., et les **balises** (§6.4) permettent de connaître l'état réel du système.

Nous avons mis en place un **historique** (§10) pour les mesures, les commandes effectuées par l'opérateur et divers messages d'alarme. Ces données sont systématiquement archivées dans la base O_2 et peuvent être visualisées sur l'écran à la demande de l'opérateur. L'état de la machine avant et après une défaillance (panne, alarme, mauvaise manipulation, etc.) est ainsi mémorisé.

Un **système d'alarmes** (§3.1 et §9) surveille l'évolution des mesures et signale, via les médias (haut-parleur, imprimante au fil de l'eau ou base de données), tout fonctionnement anormal du système. Des fonctions de sécurité (§3.1) gèrent les "sécurités mineures" (chapitre A, §2.4). Elles attireront l'attention de l'opérateur sur un *paramètre* précis.

12.2 Ergonomie

La redéfinition du mode de communication entre les *concentrateurs* et les écrans graphiques (§6) a joué un rôle essentiel sur l'ergonomie du système :

- **La dynamique des écrans graphiques s'est considérablement améliorée** et donne une sensibilité beaucoup plus importante aux curseurs. Ces derniers ont été entièrement refaits pour les rendre plus fonctionnels (réglage du pas, par exemple (§6.1.1)), tout en gardant le même aspect graphique. Les lectures sont plus vives et l'envoi des commandes vers les *concentrateurs* (§1.3) n'est plus perturbé.
- **Les écrans graphiques reflètent l'état réel du système** à un instant donné (§2.2).
- **L'action d'envoyer une commande** (bouton enfoncé, curseur déplacé, ...) **est répercutée et mémorisée sur tous les écrans jumeaux** ouverts (ou à ouvrir) grâce aux formulaires.
- Ces deux derniers points permettent de reprendre un réglage sur un autre poste de contrôle.
- GMS n'impose plus sa fréquence d'échantillonnage sur les mesures qui sont maintenant remontées sur décision des *concentrateurs*. Nous avons ainsi une **meilleure perception sur l'évolution temporelle des mesures**.
- L'opérateur n'a plus à gérer la connexion écran-concentrateur.

12.3 Fiabilité

En ce qui concerne la fiabilité, les deux paramètres à considérer sont la fréquence des défaillances et leurs conséquences sur le fonctionnement du système (chapitre A, §2.4). Nous avons minimisé l'effet des défaillances en rendant les *concentrateurs* autonomes et indépendants. Les pannes sont tolérées et le mauvais comportement d'un *concentrateur* (perte de liaison, non fonctionnement, ...) ne nuit en rien aux fonctions dépendant des autres *concentrateurs*.

Le **fonctionnement en mode dégradé** assure la continuité du système. L'emploi du langage de 4ème génération, O_2C (chapitre B, §3.3.1), donne de la **robustesse au code**. Il gère la mémoire et permet l'utilisation de types complexes ce qui entraîne une meilleure lisibilité du code.

12.4 Maintenance

Nous avons voulu donner une orientation objet à notre système informatique (§2.5) pour bénéficier des avantages de la technologie objet : maintenance, réutilisation du code et portabilité du système facilitées. **Le système devient plus modulaire** et sa cohérence est garanti par le SGBDOO O_2.

La présence des fonctions associées aux *paramètres* justifie le choix de la **génération automatique ou assistée du code** par le SGBDOO O_2. En effet, chaque *paramètre* a sa propre fonction qui, la plupart du temps, garde le même algorithme. Imaginons le travail d'édition de code que demanderait un système gérant plus d'un millier de *paramètres*... De plus, l'unicité du *paramètre* peut être vérifiée si elle se trouve dans la base.

Le générateur de code (§11.2) a demandé un travail de développement assez long. Mais cet effort sera rentabilisé à court terme, quand l'utilisateur voudra intégrer de nouveaux *paramètres* ou modifier l'ergonomie des écrans, il se laissera guider par l'environnement de travail créé dans O_2. **Le délai de développement sera ainsi raccourci**.

12.5 Conclusions

Le mécanisme de génération de code par le SGBDOO O_2 nous offre un gain en temps important lors de la génération d'une application. Ce gain est difficile à chiffrer car le temps de développement peut varier considérablement suivant la complexité des objets graphiques ou du code spécifique à éditer sur O_2 (§11.1), Nous pouvons prendre l'exemple du banc de test pour la courroie composé de :

- un concentrateur ;
- deux cartes d'interface ;
- 64 canaux physiques ;
- 31 mesures et autant de fonctions associées ;
- 39 commandes et autant de fonctions associées ;
- un écran graphique ;
- 30 sous-modèles (§7.5).

La figure C.45a chiffre en heures le temps de développement consommé pour réaliser ce banc de test. Le générateur de code nous a permis de faire le développement en une journée au lieu d'une semaine. C'est l'édition de l'application GMS qui demandait le plus de temps ; or, actuellement, l'édition est entièrement automatique ce qui nous offre un gain de temps "énorme".

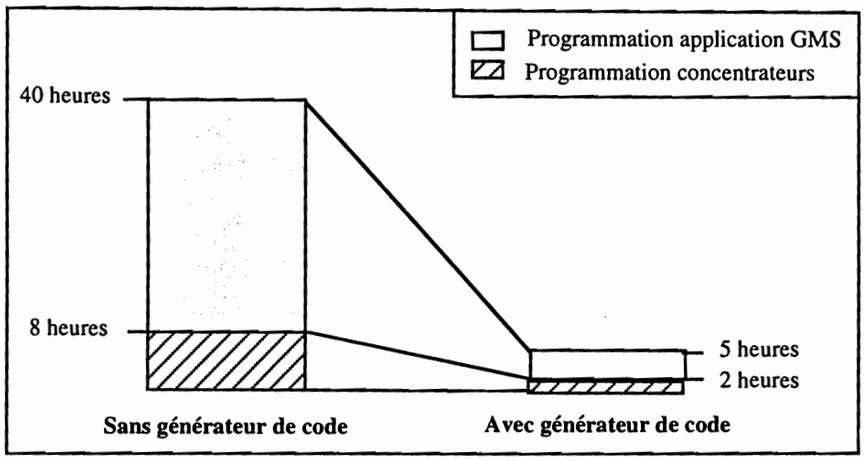


Figure C.45a - Temps de développement pour un banc de test

La figure C.45b nous donne les effectifs, en hommes/années, utilisés pour le développement de chaque module du projet Vivitron. Le développement de l'ensemble du projet a demandé **5 hommes/année.**

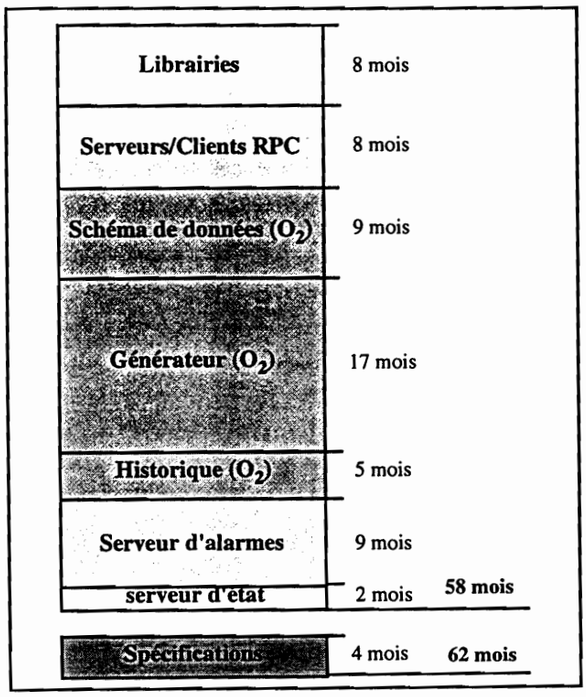


Figure C.45b - Temps de développement pour le projet Vivitron

Le tableau T.10 donne à titre indicatif le nombre de données rentrées dans la base jusqu'à ce jour.

	Quantité rentrée dans la base	Quantité remplie
Equipements atomiques	132	40%
Equipements composites	23	30%
***Concentrateurs* (+ test)**	18	100%
Applications GMS (+ test)	11	50%
Mesures installées	384	30%
Commandes installées	180	30%
Canaux VME connectés à un ou plusieurs *paramètres*	490	30%
Canaux série connectés à un ou plusieurs *paramètres*	96	60%

Tableau T.10 - Remplissage de la base au 1er janvier 1996

La figure C.45c illustre le bilan des lignes de code générées par la base O_2 ou écrites par le développeur et le tableau T.11 donne le nombre de lignes. Autour de **80 000 lignes de code** (~ 70 lignes/jour) ont été écrites par les développeurs pour réaliser le noyau de l'architecture informatique de contrôle et commande. Le double de lignes ont été générées automatiquement par la base O_2. Il nous reste aujourd'hui un certain pourcentage de code à écrire ou à générer. Tout ce qui concerne GMS sera généré automatiquement au fur et à mesure que les nouveaux écrans graphiques seront spécifiés. Le code *concentrateur* sera édité en partie à la main, en ce qui concerne les fonctions des *paramètres*.

	Langage C		Langage O_2C		Quantité de code généré aujourd'hui
	écrit	généré	écrit	généré	
Librairies, serveurs	41 800	--	--	--	100%
Schéma O_2, données et générateur	--	--	41 000	49 000	80%
Schéma O_2, historique	--	--	6200	--	90%
Code concentrateur	--	54 000	--	--	50%
Code GMS	--	52 000	--	--	50%
Lignes de code écrites	83 600				
Lignes de code généré	155 000				

Tableau T.11 - Lignes de code écrites et générées

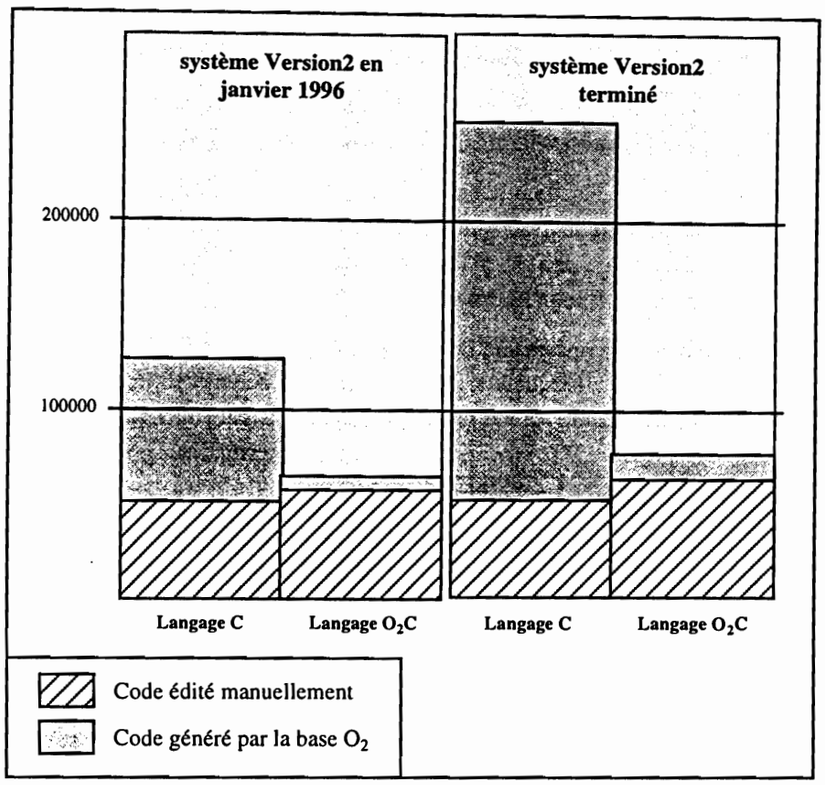


Figure C.45c - Bilan des lignes de code

Enfin, le tableau T.12 donne une idée de la grandeur à ce jour des deux schémas d'objets qui ont été créés pour modéliser le système de contrôle et commande et de l'historique dans la base de données O_2.

	Nombre de classes	Nombre de méthodes
Version2	148	2047
Historique	22	100

Tableau T.12 - Dimension des schémas d'objets Version2 et Historique

Le nombre de classes et de méthodes est cité ici à titre indicatif, car la base ne cesse de s'agrandir au fur et à mesure que de nouveaux équipements sont insérés dans notre système informatique.

Conclusion

Aujourd'hui, le nouveau système informatique de contrôle et commande est opérationnel sur certaines parties de l'accélérateur Vivitron. Il devra, à terme, gérer plus de 1500 mesures et commandes, et fournir aux opérateurs une interface de contrôle et commande la plus conviviale possible.

Des outils logiciels de pointe très novateurs nous ont permis de réaliser un système de contrôle et commande très original :

- VxWorks : exécutif temps réel ;
- SL-GMS : générateur d'interfaces graphiques orienté objet ;
- O2 : Système de Gestion de Base de Données (SGBD) orienté objet ;

Ces logiciels sont implantés sur des stations de travail, intégrant pleinement le système d'exploitation UNIX, et sur des ordinateurs frontaux, le tout constituant une architecture distribuée sur un réseau Ethernet. Dans un premier temps, les processus clients et les processus serveurs échangeaient des informations exclusivement par l'exécution de procédures à distance (RPC). Mais, la remontée systématique et périodique des mesures occasionnait des saturations sur les affichages graphiques des stations de travail. Ces blocages pouvaient perturber grandement l'envoi de commandes.

Nous avons introduit dans notre système le protocole X comme support de communication entre les écrans graphiques et les ordinateurs frontaux. Cette nouveauté nous a permis d'améliorer considérablement la dynamique du système. Le traitement des mesures est ainsi distribué sur les ordinateurs frontaux et les résultats sont concentrés et affichés sur les écrans graphiques. Notre architecture informatique devient plus fonctionnelle et plus robuste.

Pour garantir la cohérence entre tous les modules, la description physique et fonctionnelle de chaque ordinateur frontal ainsi que le comportement graphique de chaque écran GMS ont été introduits dans la base de données O_2 sous forme d'un schéma d'objets. L'unicité et la cohérence des paramètres sont ainsi garanties et le risque d'introduction d'une éventuelle erreur, lors d'une modification, diminue considérablement.

La description des équipements est toujours associée aux programmes les gérant. Comme cette description est intégrée dans la base de données, nous avons choisi de faire générer automatiquement le code répétitif par le SGBDOO O_2. Des outils lui ont été ajoutés pour faciliter l'édition du code non répétitif en liaison avec les informations contenues dans la base.

Le rôle de notre SGBDOO est, outre la description, le stockage, l'accès et la maintenance de grandes quantités de données, la génération automatique du code. Cette originalité rend la maintenance accessible à toute personne non spécialisée. La base devient l'élément central du système de contrôle et commande.

L'utilisation d'outils logiciels orientés objet (O_2 et SL-GMS) nous a permis de fonder la construction de notre architecture informatique sur les concepts objets. La programmation par objets nous permet d'avoir un système plus modulaire, d'améliorer sa fiabilité, de réutiliser les modules et de réduire considérablement la charge de développement liée à toute modification des équipements Vivitron.

Les développements effectués lors de ce projet pourront être adaptés au contrôle et à la commande des équipements scientifiques associés au Vivitron, tels que les multidétecteurs ICARE ou EUROGAM phase II.

Bibliographie

[1] J.R. Lutz, J.C. Marsaudon – *Modern tandem control systems* ;
EA & EB '92, Padova (Italie), Nucl. Instr. Meth., A4022, 1993.

[2] D. A. Bromley – *The development of electrostatic accelerators* ;
Nucl. Instr. Meth., 122, pp 1-34, 1974.

[3] P. J. Bryant – *A brief history and review of accelerators* ;
CERN, Geneva, Switzerland, pp 1- 2.

[4] R. J. Van de Graaff – *Tandem electrostatic accelerators* ;
Nucl. Instr. Meth., 8, pp 195-202, 1960.

[5] A. Nadji – *Les faisceaux du Vivitron* ;
Thèse, CRN Strasbourg, 1989.

[6] J.M. Helleboid – *Un système de charge pour le Vivitron* ;
CRN-VIV-49, 1987.

[7] *Le Vivitron un nouvel accélérateur électrostatique* ;
CRN-VIV-12, 1983.

[8] J.R. Lutz, J.C. Marsaudon – *The vivitron process control* ;
ICALEPCS'89, Vancouver (Canada), Nuclear Inst and Meth, 1990.

[9] M. Roumier – *Etude des phénomènes transitoires dans le Vivitron lors d'une décharge électrique*, CRN, Strasbourg, 1994.

[10] J.R. Lutz, Ch. Muller, D.Stadelmann – *Rapport d'activité* ;
CRN Strasbourg, p. 135, 1987.

[11] A. Dorseuil, P. Pillot – *Le temps réel en milieu industriel* ;
Editions Dunod, Paris, p.54, 1991.

[12] J. Henshall, S.Shaw – *OSI. Les normes de communication entre systèmes ouverts* ;
Editions Masson, Paris, 1991.

[13] J. Postel – *Internet Protocol, Darpa Internet Program Protocol Specification* ;
RFC 791, septembre 1981.

[14] J. Postel – *Internet Control Message Protocol, Darpa Internet Program Protocol*
Specification, RFC 792, septembre 1981.

[15] J. Postel – *Transmission Control Protocol, Darpa Internet Program Protocol*
Specification, RFC 793, septembre 1981.

[16] J. Reynolds, J. Postel – *Assigned numbers* ;
RFC 1060, mars 1990.

[17] J. Postel – *User Datagram Protocol, Darpa Internet Program Protocol Specification* ;
RFC 768, août 1980.

[18] A. Nye – *Volume 0, X Protocol Reference Manual for X Version 11* ;
O'reilly & Associates, Inc, juillet 1989.

[19] *Network File System Protocol Specification* ;
Sun microsystems, 1989.

[20] *Network Programming Guide* ;
Sun microsystems, mars 1990.

[21] *The VMEbus Specification, Conforms to : ANSI/IEEE STD1014-1987, IEC 821 and 297* ;
VMEbus International Trade Association (VITA), 1987.

[22] *VxWorks Programmer's Guide, Release 5.1* ;
Wind River Systems, février 1993.

[23] D.R.C. Hill – *Analyse orientée objet & modélisation par simulation* ;
Masson, Paris, 1991.

[24] M. Adiba, C. Collet – *Objets et bases de données, le SGBD O_2* ;
Hermès, Paris, 1993.

[25] J.J.Aubert, P. Dixneuf – *Conception et programmation par objets* ;
Masson, Paris, pp.122, 1991

[26] *The O_2 User Manual*, Version 4.5 ;
O_2Technology, novembre 1994.

[27] *SL-GMS, Reference Manual* ;
Sherrill-Lubinski Corporation, août 1993.

[28] *VDAC12-16, Version 2.x, Isolated Analog Output Module Manual* ;
Oettle & Reichler Industrial Computers, septembre 1993.

[29] *VADC2x, Version 1.x, 12 Bit Analog to Digital Converter Manual* ;
Oettle & Reichler Industrial Computers, septembre 1993.

[30] *VMOD, Modular Input/Output Board for the VMEbus, User's Manual* ;
PEP Modularl Computers, 1989.

[31] *XYCOM, Counter Module with Quadrature* ;
1988.

[32] *Interface TCS 1001, Operating Instructions* ;
Balzers Pfeiffer, 1993.

[33] *TPG300, Operating Instructions* ;
Balzers Pfeiffer, décembre 1990.

[34] *INCAA* ;
Incaa Computers, mai 1992

[35] *DTM 130 Digital Teslameters, Users's Manual, V 5.0*
Group 3 Technololgy.

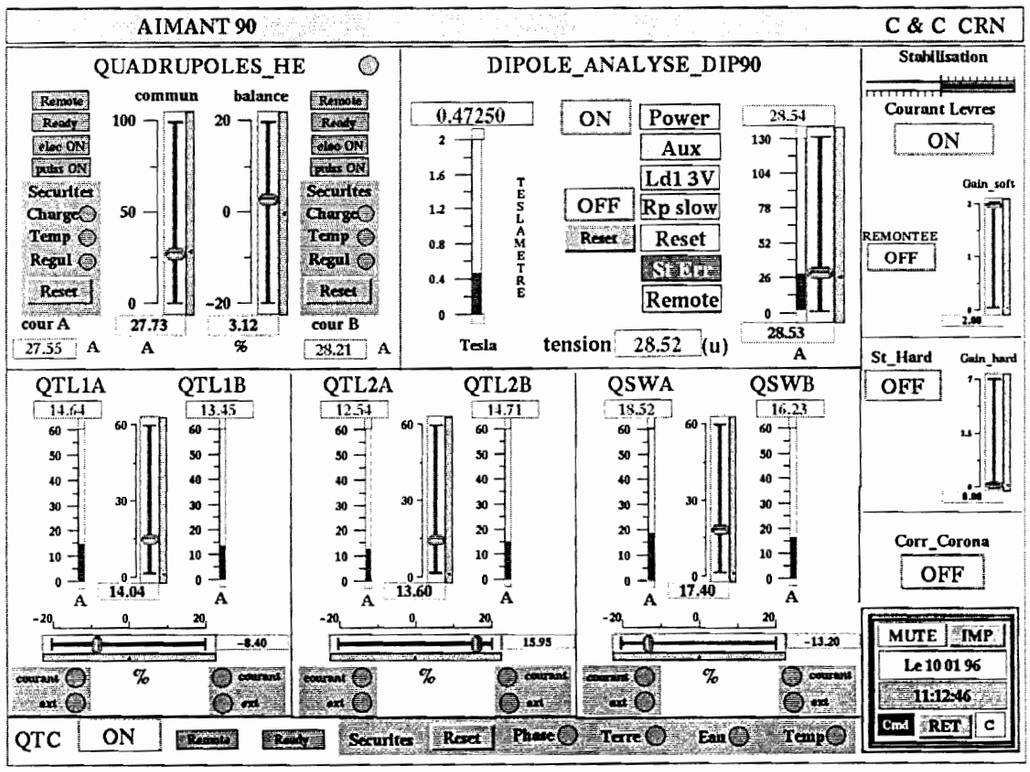
[36] R. Oakley, M.C. Liber – *Les systèmes embarqués ont leur fenêtrage* ;
Electronique, numéro 52, pp. 78, octobre 1995.

[37] G.Booch – *Object Oriented Design with Applications* ;
Benjamin & Cummings, New York, USA, 1991.

[38] *VxWorks Board Support Documentation, Motorola MVME 167* ;
Win River Systems, août 1991.

Annexe I L'Ecran Graphique de l'Aimant 90

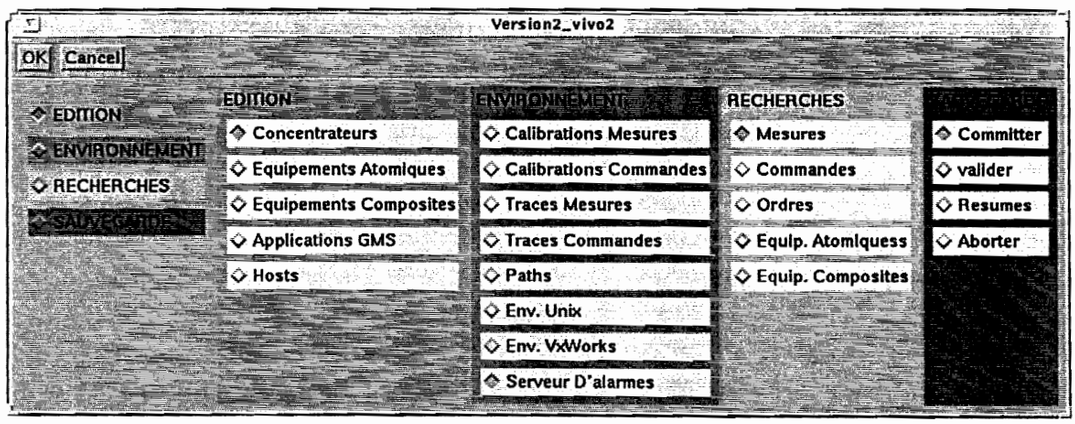
Cet écran a été réalisé avec GMS pour contrôler l'aimant d'analyse 90° situé à la sortie de l'accélérateur Vivitron (chapitre A, §1.2.2). Il est composé de 8 *sous-modèles* (chapitre C, §7.5). Les 3 *sous-modèles* en bas de l'écran à gauche sont composés chacun de deux quadripôles couplés (QTL1A/QTL1B, QTL2A/QTL2B et QSWA/QSWB). Pour chaque *sous-modèle* le curseur vertical règle les alimentations des deux dipôles simultanément, et le curseur horizontal règle un décalage de ± 20% entre les deux alimentations. Les liens entre un *sous-modèle* et les canaux physiques sont entièrement gérés par la base de données O_2.



Annexe II L'Interface Homme Machine O2

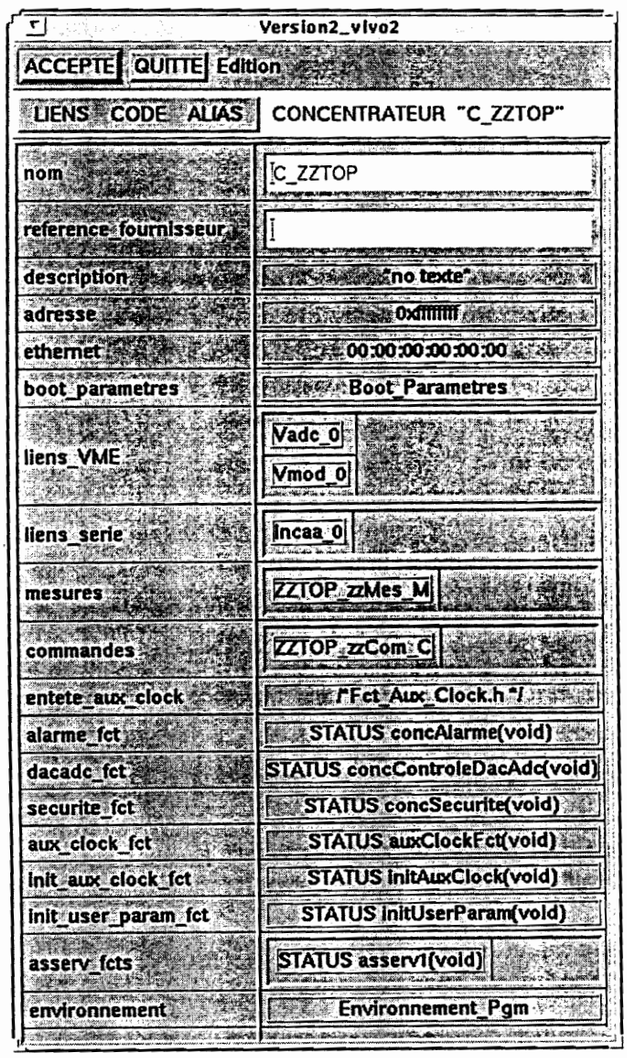
Cet interface a été entièrement réalisée avec le générateur d'interfaces graphiques intégré dans le produit O_2. L'image ci-dessous montre le **tableau de bord** du système informatique de contrôle et commande qui se compose de 4 menus :

- **Edition**
  Les *concentrateurs* (chapitre B, §4.2.1), les *équipements composites* et les *équipements atomiques* (chapitre C, §7.1), les écrans GMS (chapitre C, §7.5) et les stations de travail hôtes sont modélisés dans la base sous forme d'objets. Ce menu permet de créer et de modifier ces objets.
- **Environnement**
  Un environnement de travail est nécessaire pour générer le code des *concentrateurs* ou des applications GMS (valeurs par défaut, chemins d'accès aux répertoires, options de compilation...). Cet environnement est modélisé sous forme d'objets et peut être configuré dans ce menu.
- **Recherches**
  Ce menu permet de rechercher dans la base les mesures, les commandes, les *ordres* (chapitre C, §2.6) les *équipements atomiques* ou les *équipements composites*, par des clés spécifiques (recherche d'une mesure par le nom de son *concentrateur*, recherche d'un *ordre* par le nom de sa mesure...).
- **Sauvegarde**
  Ce menu permet de sauvegarder dans la base toutes les modifications réalisées sur les objets (boutons **Commiter** et **Valider**) ou de sauvegarder sur des fichiers *o2resume* (chapitre C, §1.4.2) toutes les données relatives aux mesures et aux commandes de chaque concentrateur (bouton **Resumes**).



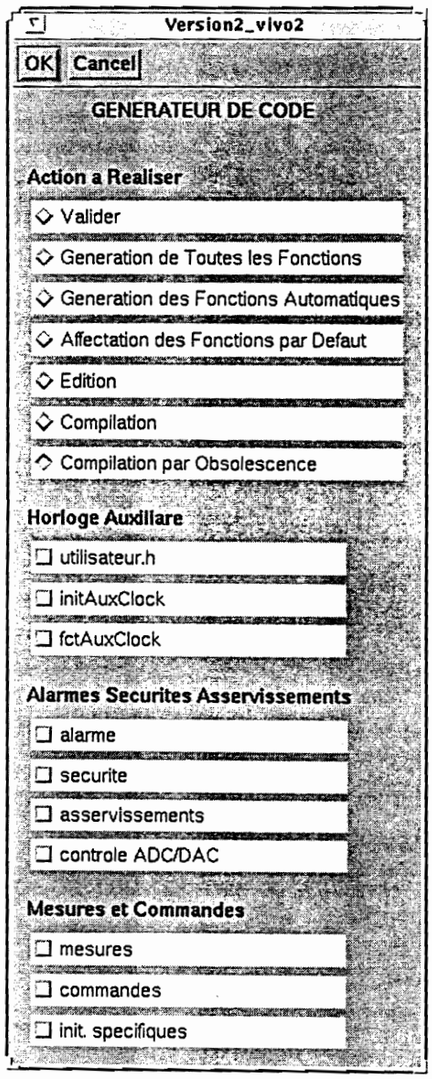
Annexe III L'Objet Concentrateur

Nous avons ici un exemple d'objet ***Concentrateur*** tel qu'il est représenté dans l'IHM O_2. Les *liens VME*, les *liens série* (chapitre C, §5.1), les commandes et les mesures sont représentés sous forme de liste d'objets. Les fonctions codées manuellement (chapitre C, §11.3) forment des objets (alarme_fct, securite_fct...) qui contiennent un éditeur de texte pour éditer les fonctions. L'objet ***Concentrateur*** dispose de trois menus (**Liens**, **Code** et **Alias**) qui permettent de créer ou supprimer des *liens*, de générer le code du *concentrateur* et de manipuler les *alias* (chapitre C, §11.3.1).



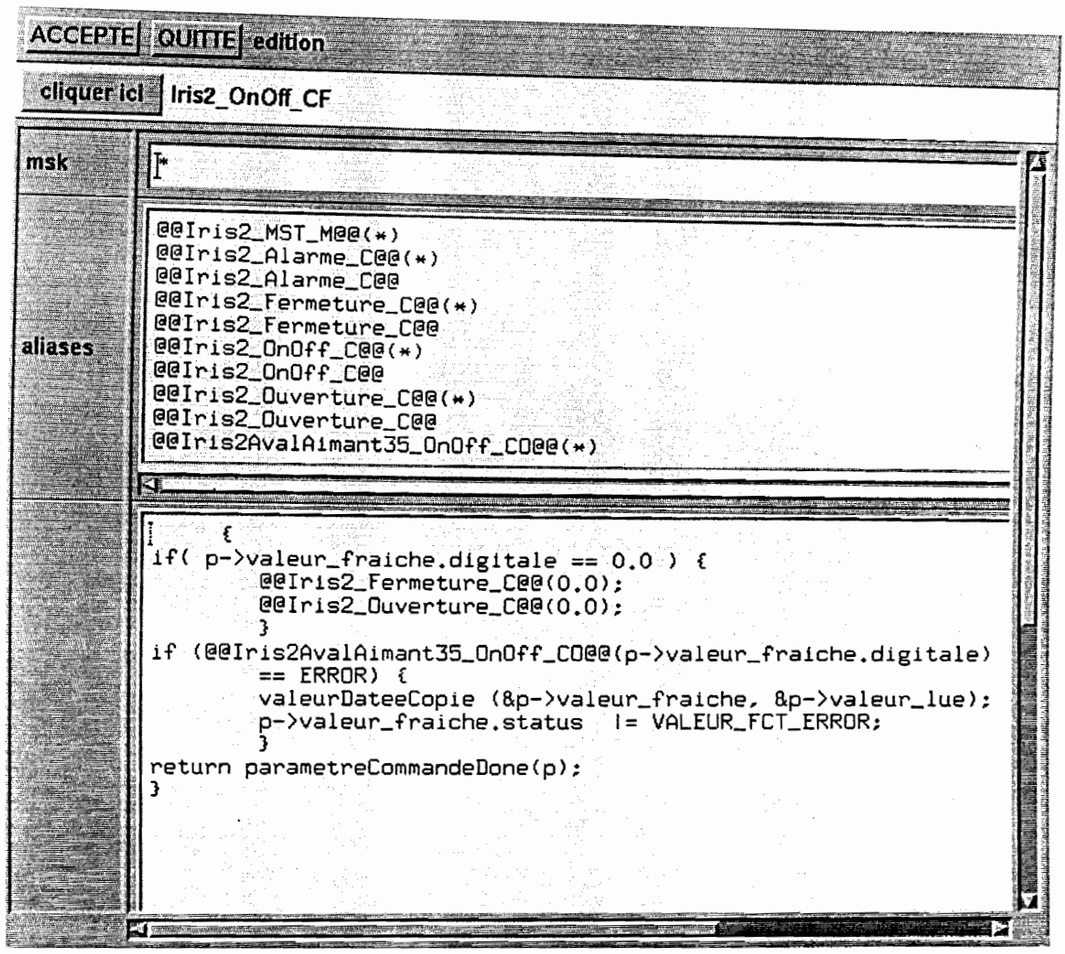
Annexe IV Le Générateur De Code

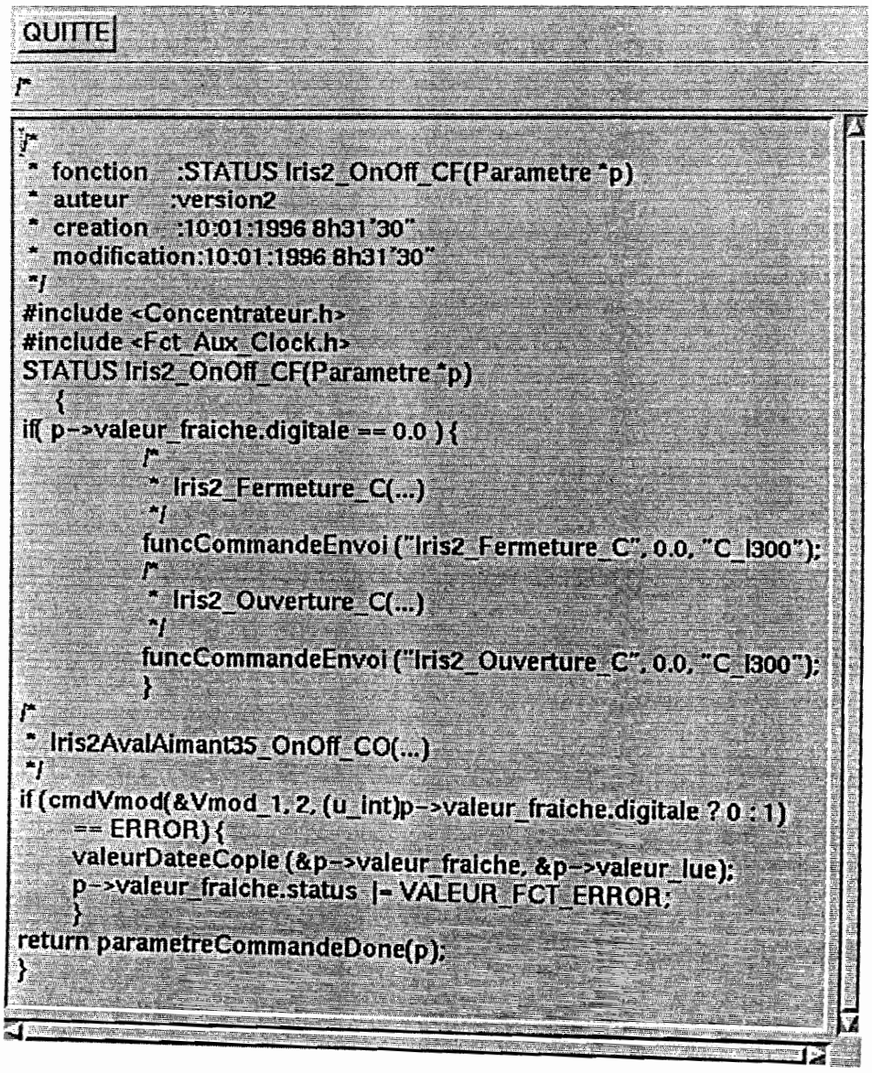
Le menu du générateur de code d'un concentrateur a l'aspect de l'image ci-dessous. Il se décompose en deux parties : lés actions à réaliser et la liste des fonctions à générer. Nous avons la possibilité de générer (éditer et compiler) toutes les fonctions du concentrateur en une seule action, ou d'éditer et compiler que des fonctions spécifiques (alarme, mesures...)



Annexe V Les Alias

La première image montre l'édition d'une fonction de commande qui réalise un On/Off sur un iris. L'utilisateur ne manipule ici que des symboles (des *alias* (chapitre C, §11.3.1)) correspondant aux actions à réaliser. Les accès aux mesures et aux commandes (nom de la commande, numéro de canal...) sont transparents. Un pré-compilateur remplacera les *alias* par les appels aux fonctions adéquates et donnera le texte source de la deuxième image. La liste des *alias* accessibles par la fonction apparaît dans le champ **aliases**.





QUITTE

```
/*
 * fonction   :STATUS Iris2_OnOff_CF(Parametre *p)
 * auteur     :version2
 * creation   :10:01:1996 8h31'30"
 * modification:10:01:1996 8h31'30"
 */
#include <Concentrateur.h>
#include <Fct_Aux_Clock.h>
STATUS Iris2_OnOff_CF(Parametre *p)
    {
if( p->valeur_fraiche.digitale == 0.0 ){
            /*
             * Iris2_Fermeture_C(...)
             */
            funcCommandeEnvoi ("Iris2_Fermeture_C", 0.0, "C_I300");
            /*
             * Iris2_Ouverture_C(...)
             */
            funcCommandeEnvoi ("Iris2_Ouverture_C", 0.0, "C_I300");
            }
/*
 * Iris2AvalAimant35_OnOff_CO(...)
 */
if (cmdVmod(&Vmod_1.2, (u_int)p->valeur_fraiche.digitale ? 0 : 1)
    == ERROR){
    valeurDateeCople (&p->valeur_fraiche, &p->valeur_lue);
    p->valeur_fraiche.status |= VALEUR_FCT_ERROR;
    }
return parametreCommandeDone(p);
}
```