Rapport de Stage 1999/2000

LEAU Maxime

Unimédia Communication,
Université d'Angers, Département d'Informatique

DESS d'Informatique.

Septembre 2000

Table des matières

Table des figures

Chapitre 1

Introduction

Le stage dont ce document est le rapport final, s'inscrit dans la formation du **DESS d'Informatique** que j'ai suivi à l'Université d'Angers, au département d'Informatique. La durée de ce stage est de cinq mois. Il a lieu en fin de formation, après cinq mois de cours. J'ai effectué ce stage dans la société **Unimédia Communication**.

Cette première partie du rapport à pour but de donner une brève description des activités de la société qui m'a accueilli en stage, puis une brève description des différents travaux que j'ai effectué pour cette société.

1.1 Unimédia Communication

La société **Unimédia Communication** a été crée en 1995 par Mr **Jean-Pierre Bretaudière**, à Angers. Cette société revendique le statut de "**start-up**". Elle se veut être un incubateur d'idées, une société de développement de projets. En l'occurence, il s'agit de développer des services internet, directement pour le public, ou pour différents fournisseurs de ces services.

De cette idée, deux principaux projets ont abouti :

- **Angers Journal**, un journal local en ligne,
- **YouplaBourse**, un site d'information boursière en ligne.

Ces deux projets sont désormais indépendants.

1.2 Angers Journal

Angers Journal est un journal angevin d'information local en ligne.

L'équipe qui y travaille est essentiellement composée de journalistes, formant une rédaction complète. Ils sont la base du journal, dans la mesure ou ils en fournissent le contenu, c'est à dire les articles et les photos. Une nouvelle "une" est réalisée chaque jour, et est disponible en ligne le soir même, vers 19h.

L'originalité du projet réside essentiellement dans sont "contenant", c'est à dire le support électronique du journal, entretenu et développé en collaboration avec une équipe particulière.

Les articles et les photos réalisés sont stockés dans une base de données, et présentés sous forme de pages web générées par un serveur.

Le tout est disponible au public à l'adresse internet suivante :

`http ://www.angersjournal.fr`

1.3 Youplabourse

Youplabourse est un site d'information boursière en ligne.

Le but de ce site est de présenter, en temps réel[1], les cours des différentes valeurs côtées et des différents indices, à la **Bourse de Paris**, et sur les plus grandes places boursières internationnales pour les principaux indices. Le site présente aussi des dépêches, d'ordre économique, se rapportant à ces différentes cotations.

1.3.1 Acquisition des informations

Les cours des ces différentes cotations et de ces différents indices, arrivent à Angers via un satellite. C'est la société **Standart & Poor** qui livre ces informations. Ce flux d'information est traité, et les informations sont stockées dans les bases de données du site.

Il en est de même pour les dépêches en provenance de l'agence de presse **Reuters**. Sauf que dans ce cas, une partie du traitement est effectué à Paris, dans d'autres locaux, avant que les dépêches traitées ne soient rappatriées à Angers.

D'autres dépêches, en provenance des agences de presse **PrLine**, **AGPresse**, et du **Journal des Finances**, sont traitées par différents moyens (reccupération sur des serveurs ftp chez les fournisseurs, envoie par courrier électronique...).

Enfin, une partie des dépêches présentées par le site, sont produites sur place. Il s'agit dans ce dernier cas, d'informations générées lors du décodage du flux des cotations, comme : les interruptions de cotations, les dépassements à la baisse ou à la hausse d'indicateurs sur ces cotations...

1.3.2 Présentation des informations

Une fois toutes les informations disponibles dans les bases de données, elles peuvent être présentées sous différentes formes par les serveurs web. Différents "moteurs" permettent d'extraire de ces bases de données, les informations présentées sur le site, ainsi que les différents graphiques.

Les pages **html** sont générées avec le langage **Php**, appelant lui-même les bases de données, ou les différents moteurs existant.

Le site **Youplabourse** présente la forme finale donnée à toutes ces informations à l'adresse suivante :

`http ://www.youplabourse.com`

Ce savoir faire a été revendu, pour le compte d'autres prestataires devant bénéficier, en leurs noms, de services similaires. Ainsi, les pages bourses des sites du **Journal des Finances**, du journal **Le Point**, du portail internet **Magéos**, du *brooker* en ligne **Boursier.com** et de quelques autres encore, sont des services mis en ligne par le site **Youplabourse**.

1.4 Travail effectué au cours du stage

Je n'ai, pour ma part, essentiellement travaillé que sur le projet relatif au site de bourse en ligne. Sous la conduite de Mr Bretaudière, j'ai activement participé au développement des outils de traitement des différents flux d'informations.

Dans cet état d'esprit, j'ai réalisé, souvent en collaboration étroite avec Mr Bretaudière, les travaux suivants :

- **MP**, une "*multiprise*" pour TCP/IP, le but de cette application étant de pouvoir redistribuer des flux vers différentes machines,

[1] En réalité avec un quart d'heure de décalage, comme nous l'impose nos fournisseurs et la **Bourse de Paris**.

- **Rts2Db**, un décodeur pour le flux d'informations de l'agence de presse **Reuters**,
- une étude, sur les moyens offerts pour la réplication avec le SGDB **MySQL**, et une mise en oeuvre de celle-ci,
- **Yalta**, un langage dédié à l'analyse technique boursière.

Outre ces travaux, j'ai souvent participé à l'administration des machines **Linux** utilisées par le site, installant de nouveaux services ou de nouveaux systèmes sur celles-ci.

Chapitre 2

Multiprise TCP/IP

2.1 Problématique

Unimédia reçoit, en direct, les cotations des différentes valeurs côtées à la Bourse de Paris. Ce *flux* d'informations arrive par satellite à Angers.

Le signal satellite est décodé par une machine dédiée à cette tâche, nommée **CSP**. Unimédia dispose de deux **CSP** à Angers. Ces machines décodent donc le signal satellite et réémettent le flux décodé, sur le réseau local de l'entreprise, via le protocole **TCP/IP**, sur le port 6010. Malheureusement, ces machines n'acceptent qu'une connexion à la fois. Or, pour des raisons de sécurité et de performances, il est préférable que plusieurs bases de données puissent être alimentées par le même *flux*, et ceci depuis des machines différentes.

2.2 Etude de solution

C'est, partant d'une solution existante, que la *Multiprise TCP/IP*[1] a été dérivée.

2.2.1 Solution existante

Un logiciel, le *msx*, existe déjà pour effectuer cette tâche. Il remplit toutes les fonctionnalités désirées :

- Connexion et reprise de connexion en cas d'erreur, avec le **CSP**,
- Découpage du *flux* en paquets correspondant à un élément d'information,
- Gestion de connexions multiples de clients distants.

Malheureusement, l'implémentation choisie pour *msx*, le rend, intrinsèquement, peu performant, et il est, du fait d'une longue évolution de différentes versions plus ou moins spécialisées, boguées.

"*msx*" a été écrit en langage **C** "natif", avec un système d'attente sur les *sockets*[2] ouvert en entrée et en sortie. De telle sorte que des évènements qui peuvent être traités indépendemment les uns des autres, sont traités par *msx* de manière successive.

C'est une implémentation *mono-thread* classique, c'est à dire un seul programme s'exécutant sans concurrents sur ses ressources.

[1] *Multiprise TCP/IP* ou *mp*.
[2] Les *sockets* ou port de communication de **TCP/IP**.

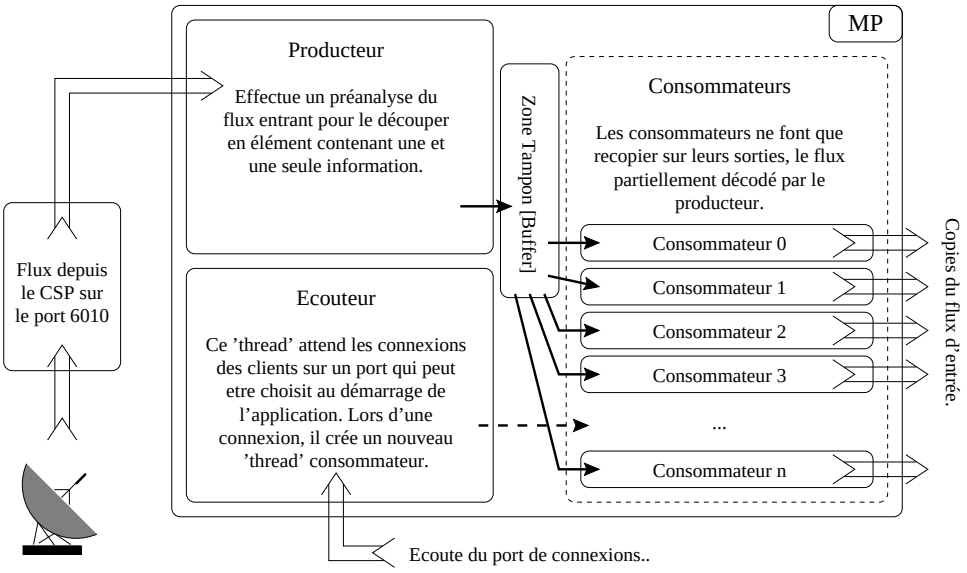

FIG. 2.1 – MP - Principe.

2.2.2 Solution adoptée

Toutes les grandes lignes de *msx* sont réutilisables. Il faut simplement trouver la manière de faire s'exécuter *parallèlement* les différents services qu'il met à disposition.

Une première idée a été de réécrire *msx* en langage **Java**, comme mes connaissances en programmation me le permettaient. En effet, je sais faire avec **Java** tout ce qui me semble nécessaire pour une telle application, à savoir : programmer des *threads*, et programmer **TCP/IP**. Malheureusement, la garantie de performance fournie par **Java** est tout à fait moindre par rapport à celle que nous fournie un programme écrit en **C**.

Le programme a donc été écrit en **C**, à l'aide des bibliothèques pour la communication TCP/IP et de la bibliothèque "`pthread`" sous Linux.

2.2.2.1 Principe général

L'application se décompose en trois sous-programmes distincts fonctionnant chacun dans des threads différents. Les dénominations que j'ai choisi sont les suivantes : `producteur`, `consommateurs`, `écouteur`. La figure 2.1 donne le principe général de cette application.

Tout le problème a été de trouver le moyen pour synchroniser ces différents éléments, concurrents sur la zone tampon, pour, en particulier, que les consommateurs ne puissent pas lire le buffer du flux, pendant que celui-ci est mis à jour par le producteur. Et réciproquement que le producteur ne tente cette mise à jour, tant que les consommateurs n'ont pas fini de lire ce buffer.

2.2.2.2 Utilisation des verrous

La bibliothèque "`pthread`" met à la disposition du programmeur des outils, non seulement pour la création, dans un même processus, de *thread* exécutant une portion particulière de code, mais aussi pour la création de verrou ou *mutex*[3]. Ces

[3] *mutex* pour "*MUTual EXclusion device*".

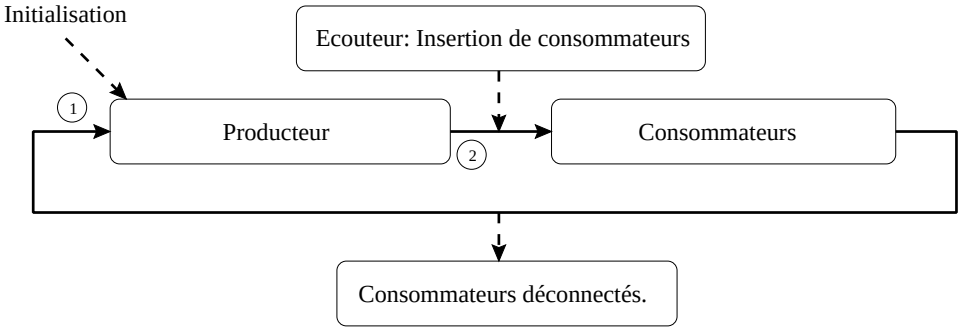

FIG. 2.2 – MP - Le cycles de MP.

mutex permettent de protéger les données, partagées par les différents *threads*, des accès concurrents.

Dans cette application, la variable en accès concurrent est le buffer dans lequel le producteur place les éléments d'informations qu'il laisse ensuite à la disposition des clients. Le cycle d'exécution est illustré par la figure 2.2.

Un couple de *mutex,* "*out*" et "*in*", est mis à la disposition de chaque consommateurs se connectant. Ces *mutex* sont utilisés ainsi :

1. Les consommateurs libèrent leurs *mutex out*, permettant ainsi au producteur de reprendre la main.
   Le producteur prend possession des *mutex out*.
   La lecture dans le flux et la mise à jour du buffer est faite.
2. Le producteur libère les *mutex in*, permettant ainsi aux consommateurs de reprendre la main.
   Les clients prennent possession des *mutex in*.
   Chaque consommateur envoie une copie du flux aux clients connectés.
3. Retour en 1.

L'insertion de nouveaux consommateurs est gérée elle aussi par un *mutex* particulier. Ce *mutex* est libéré par le producteur, au début de la phase 2, puis réapproprié immédiatement. De cette façon, si un client s'est manifesté auprès de l'écouteur, il peut être inséré dans le cycle à ce moment précis, le *mutex* étant acquis par le premier *thread* l'ayant sollicité.

2.2.2.3 Lecture et transformations du flux

Le flux d'entrée donne les informations entre des drapeaux particuliers, des identifiants. Ces identifiants décrivent la provenance de l'information : `0x17` pour la Bourse de Paris, `0x1F` pour l'heure système... Dans tous les cas, ces identifiants sont caractérisés par le fait que :

$$\texttt{ID} \leq \texttt{0x20}$$

On trouve donc dans le flux, des *trames* brutes de la forme :

`[ID]...[17]rPCACt611585[ID]...[1F]15 :48[ID]...`

La décomposition opérée par *msx* et désormais par *mp* permet d'encapsuler complètement ces informations en les terminant par un symbole particulier `EOT` (`EOT = 0x03`). De plus, il faut pour un traitement futur, ajouter aux trames en `0x1F`, la date courante.

On obtient donc des trames précédentes, les nouvelles trames :

```
[ID]...[EOT]
[17]rPCACt611585[EOT]
[ID]...[EOT]
[1F]15 :48 2000-04-18[EOT]
[ID]...[EOT]
```

"*mp*" gère donc l'insertion des symboles EOT, la détection et la modification des trames en 0x1F, et enfin le changement de date.

2.2.3 Résistance aux pannes

L'un des impératifs de fonctionnement de *mp* est la résistance aux pannes. "*mp*" doit être en mesure de résister à tous les évènements pouvant perturber le fonctionnement de la transmission du flux.

Dans cette optique, *mp* inclus les caractéristiques suivantes :

- Gestion de tous types de déconnexions de clients, gestion (depuis la version `0.06`) du blocage temporaire d'un client.
- Gestion de la reprise sur erreur lors de la lecture dans le flux (en cas d'attente trop longue).
- Gestion de la déconnexion / reconnexion au **CSP**.
- Gestion d'un fichier "`.log`" relatant tous ces événements.

Grâce à ces différentes vérifications, *mp* est capable de reprendre sur les pannes suivantes :

- Déconnexions "logicielles" ou matérielles des clients.
- Déconnexions "logicielles" ou matérielles du **CSP**, avec reprise.

2.3 Adaptation effectuée : "*mpr*"

Une adaptation de "*mp*", dédiée à un autre type de flux, a été implémentée. Il s'agit d'une version dédiée à la redistribution du flux d'informations de l'agence de presse *Reuters*.

Elle est en tout point similaire à "*mp*", à l'exception que les trames sont séparées par un autre identifiant, fixe dans ce cas. De plus, cette adaptation ne gère pas de trames d'horodatage, et par conséquent, pas de changement de date.

2.4 Evolutions possibles

"*mp*" a été développé dans une *certaine urgence*, pour pouvoir au plus vite remplacer *msx* qui n'était plus en mesure de tenir complètement ses objectifs. De ce fait, *mp* ne fait rien de plus que ce que faisait *msx*. Or, une évolution simple aurait pu être implémentée : la spécialisation des sorties...

Jusqu'à ce point, les clients qui viennent se connecter à *mp* (et antérieurement à *msx*), sont totalement *passifs*. Or, ces clients ont pour tâche de stocker dans des bases de données distinctes, les informations relatives à une place boursière particulière. Par conséquent, chaque client ignore une grande partie du flux qui lui est transmis.

En rendant les clients *actifs*, et en incluant un petit protocole de communication entre *mp* et ses clients, on doit pouvoir facilement spécialiser les sorties de *mp* pour que celui-ci n'envoie aux clients que ce qu'ils demanderons à *mp*, par exemple seulement les trames en `0x17` pour la Bourse de Paris.

Versions

"*mp*" s'est décliné, pendant son développement, en plusieurs versions présentant ces caractéristiques :

- `0.00` : version mettant en place le *squelette* des différents *threads,*
- `0.01` : mise en place des routines de communication (lecture / écriture dans les sockets),
- `0.02` : mise en place de fichiers de *log*, première version utilisable,
- `0.03` : correction de bugs relatifs au changement de date et à la génération du fichier de *log,*
- `0.04` : correction de bugs relatifs à l'accès exclusif du fichier de log aux différents *threads*,
- `0.05` : ajout d'un paramètre à la ligne ("`-fsp`") de commande pour activer la sauvegarde d'une copie du flux, mise en place d'une mini-documentation accompagnant les fichiers sources ("`RELEASE.notes`"),
- `0.06` : amélioration de la gestion des sockets avec les clients,
- `0.07` : augmentation de la taille du buffer des sockets en émission, (par défaut) de 30ko à 70ko (maximum autorisé par le système).

Chapitre 3

Rts2Db

3.1 Présentation du projet

La réalisation de ce projet fait suite à l'acquisition, par la société **Unimédia Communication**, du flux d'information de l'agence de presse **Reuters**.

Ces informations arrivent par satellite dans les locaux de la société à Paris. Là, une machine mise à notre disposition par la société **Reuters**, décode le flux du satellite, puis rend disponible sur le réseau local, par l'intermédiaire du protocole **TCP/IP**.

L'application *'mpr'*, dérivée elle même de *'mp'* (c.f. : 2.3 *'mpr'* page 14 et *'mp'* page 11), se connecte à cette machine et redistribue ce flux à plusieurs éventuels clients.

'rts2db' est un des clients qui vient se servir auprès de *'mpr'* pour récupérer le flux d'information **Reuters**, et le traiter.

Le traitement effectué par *'rts2db'* a pour but, de mettre des dépêches complètes dans une base de données, sous forme **html**[1].

Les nouvelles dépêches sont ensuite envoyées à Angers par **ftp**...

L'ensemble peut être vu sous la forme présentée en figure 3.1...

[1] **html** : Hyper Text Markup Language.

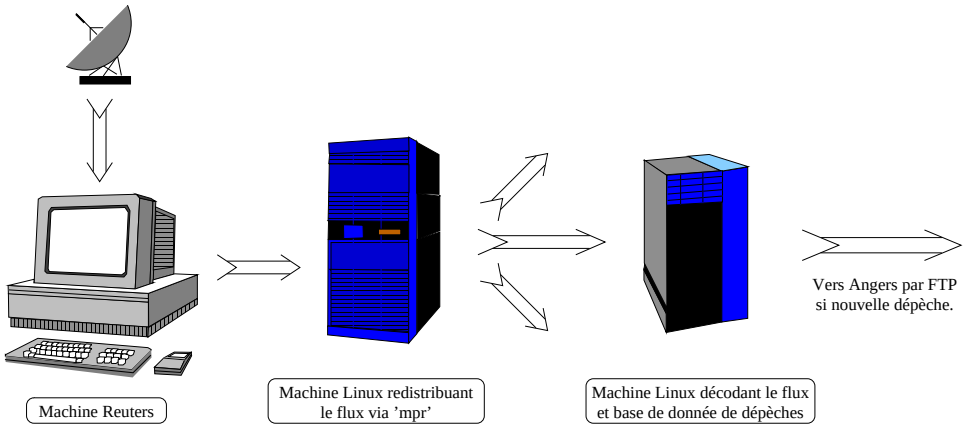

FIG. 3.1 – Rts2Db - Cheminement et traitement du flux.

Le traitement se décompose ainsi :
- détection des trames intéressantes,
- réassemblage des dépêches,
- sélection des dépêches,
- traitement des codes apparaissant dans ces dépêches et mise au format **html**,
- mise en base de données de ces dépêches.

Un dernier traitement, qui n'est pas du ressort de ce projet, consiste à rapatrier les nouvelles dépêches à Angers, et à les insérer dans la base de données courante utilisée pour le site **YouplaBourse**[2].

3.2 Fonctionnement de 'rts2db'

Le programme *'rts2db'* développé, se décompose donc en quatre parties principales. Ce programme est une réalisation commune, à laquelle je n'ai fait que participer, en compagnie de Mr. Bretaudière.

Tout le programme a été réalisé en **langage C**, les fonctions de gestions de bases de données étant effectuées par un serveur **MySQL**.

3.2.1 Détection des trames intéressantes

Le flux **Reuters** contient un grand nombre d'informations. Ce flux *"brut"*, n'est pas destiné, à l'origine, à être décodé par une autre application que celle que **Reuters** met à la disposition de ses clients pour ce faire. Or, cette application, qui ne fonctionne pas sous **Unix**, ne permet qu'une consultation locale, et sur un seul poste, des dépêches décodées.

De ce fait, outre les dépêches qui nous intéressent, passent aussi sur ce flux, des informations pour l'application dédiée au décodage du flux mise à la disposition des utilisateurs avec des menus, les cours des principaux indices des grandes places boursières, des informations sur l'abonnement du client...

Ce flux d'informations se décompose donc en trames de plusieurs types, parmi lesquelles seules certaines présentent un réel intérêt dans le décodage des dépêches.

3.2.1.1 Forme générale d'une trame

Une trame est délimitée par les caractères spéciaux suivant :

`<FS`[3]`>...<FS>`

Dans une trame, la chaîne suivant immédiatement un `<FS>`, et se terminant à un autre caractère spécial (`<US>`,`<GS>`...), donne le type de la trame. Une chaîne précédant un code `<US>` donne un descripteur de champ, la valeur suivante, entre `<US>` et `<RS>` renseigne la valeur du champ.

Ainsi, la forme générale d'une trame est :

```
<FS>TYPE<US>CHAMP<GS>VALEUR_DU_CHAMP<US>...<FS>
```

C'est le comportement *général* d'une trame, mais quelques exceptions existent.

Voiçi un exemple complet de trame :

[2] **YouplaBourse** : `http ://www.youplabourse.com`.

[3] Pour information, les codes **ASCII** des caractères spéciaux sont les suivants :

Caractère	ASCII
FS	28
GS	29
RS	30
US	31

<FS>4<RS>04478F<US>2<RS>67<RS>1<US>511<RS>235<US>nL0373864<RS>
255<US>03 JUL 2000<RS>259<US>232<RS>264<US>Italie - Ralentisse
ment de l'activité en juin -PMI<RS>456<US>B@@<RS>457<US>@@@@@@
@@@@@@@@@@@@@@@A@@@@@@@@@@@@@@@BA@@@@@@<RS>715<US>nl0000f8oT<RS>
720<US>1<RS>722<US>S<RS>725<US>RTRS<RS>749<US>FA FG DNP<RS>750
<US>IT ECI WEU EUROPE LFR RTRS<RS>752<US>FR<RS>1015<US>07 :51 :2
9<RS>1024<US>07 :17 :04<RS>1027<US>03 JUL 2000<RS>1685<US>LD<RS>
1686<US>LD_S77<FS>

3.2.1.2 Trames intéressantes

Les types de trames intéressantes à décoder dans ce flux sont au nombre de trois :

- les "*alarm news*" typées "4<RS>04478F<US>2"[4] , trames d'alarme contenant le titre des dépêches,
- les "*news*" typées "318", trames contenant le texte de dépêches à reconstituer,
- les "*remove*" typées "308", trames indiquant qu'une dépêche n'est pas passée correctement, et qu'il faut l'effacer.

Les trames "*remove*" sont passées par le système de distribution du flux, si des trames arrivent de manière incorecte, et ceci pour pallier aux défaillances possibles du système de transmission radio.

3.2.1.3 Principaux champs

Les principaux champs renseignés dans les trames qui nous intéressent sont les suivant :

Code	Description	Alarms	News	Remove
XX	Identifiant		*	*
235	Numéro de série	*		
237	Précedent		*	
238	Suivant		*	
254	Numéro de série		*	
255	Date	*	*	
256	Heure		*	
258	Texte		*	
259	Dépêche	*		
264	Titre	*		
720	Numéro d'envoie	*		
723	Tabulation		*	
725	Source	*		
749	Product Code	*		
750	Topic Code	*		
751	Valeur associée	*		
1015	Heure	*		

[4]Cette forme de typage constituant un exception à la syntaxe annoncée plus haut.

Le détail des différents champs est le suivant :
- Un *identifiant* est un numéro unique de trame. Les *identifiants* servent à réassembler les dépêches à l'aide des champs *précédent* et *suivant*,
- Un *numéro de série* indique quelle dépêche est concernée par la trame,
- La *date* et l'*heure* renseignent le moment, à l'heure GMT, où la dépêche a été produite,
- Le *texte*, dans une "*news*", est une partie du texte constituant la dépêche à réassembler,
- *Dépêche* est un simple drapeau indiquant que l'alarme concerne bien une dépêche, d'autres types d'alarmes existent mais ne nous cercernent pas,
- Le *titre* donne, dans une "*alarme*", le titre de la dépêche à venir ou à passer,
- Le *numéro d'envoie* permet de savoir si oui ou non l'alarme concernant une dépêche est déjà passée sur le flux,
- *Tabulation* est un drapeau permettant de savoir si le texte est formaté, c'est à dire sous forme de table, ou s'il ne s'agit que de texte brut,
- La *source* permet d'obtenir l'origine de la dépêche, certaines d'entre elles provenant, par exemple, non pas de l'agence **Reuters**, mais de l'agence **PRLine**,
- Les champs particuliers *product* et *topic code* contiennent des informations permettant de retrouver à quels centres d'intérêts peuvent se rapporter une dépêche, il s'agit de mots clefs, séparés par des espaces,
- Pour les dépêches d'ordre financière, le champ *valeur associée* indique quelle valeur, et sur quelle place boursière, est concernée par la dépêche.

3.2.1.4 Mécanisme adopté pour la détection

La partie décodant le flux et reconstituant les trames s'apparente à un transducteur d'états finis.

Les transitions s'effectuent sur les descripteurs de champ, et des actions associées, généralement la récupération de la valeur du champ dans une variable.

Dans le code, l'automate est décrit états par états en indiquant :
- L'état suivant, état atteint si la transition est activée,
- Un caractère de validation, si la transition s'effectue, non pas sur une chaîne, mais sur un simple caractère,
- Une chaîne de validation, si la transition s'effectue sur une chaîne,
- Un caractère final, qui détermine sur quel caractère la chaîne doit se terminer, si la transition s'effectue sur une chaîne,
- Un pointeur vers une fonction, qui correspond à l'action à effectuer si la transition est activée,
- Un pointeur vers une variable, qui correspond, dans une structure, à la valeur du champ à traiter quand il y en a un.

La figure 3.2 montre un schéma, légèrement simplifié, du transducteur utilisé pour le décodage de ce flux en trames.

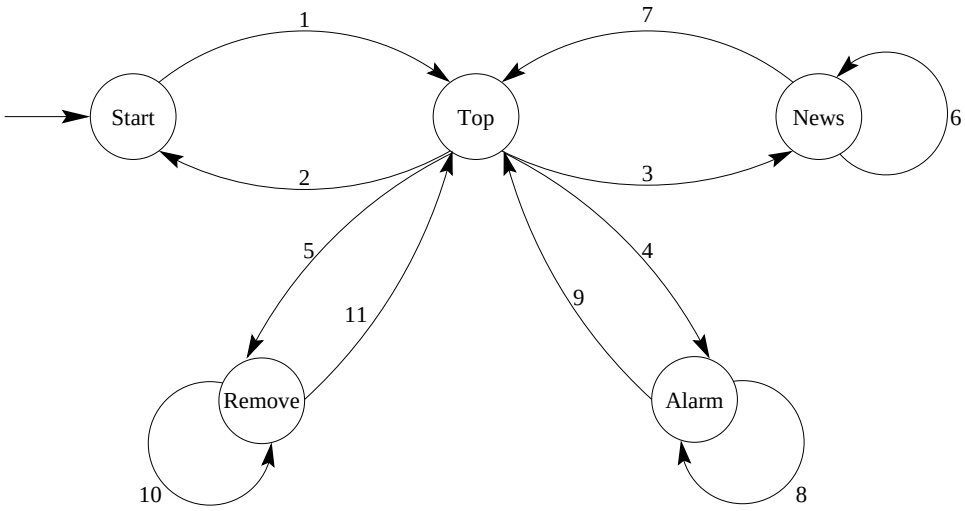

FIG. 3.2 – Rts2Db - Schéma simplifié du transducteur utilisé.

Voiçi le détail de quelques unes des transitions de ce transducteur :

n°	Etat initial	Etat final	Caractère	Chaîne	Fin de chaîne	Action	Variable
1	Start	Top	<FS>				
2	Top	Start	*				
3	Top	News		"318"	<US>	News trouvée	
4	Top	Alarm	(...)				
5	Top	Remove		"308"	<US>	Remove trouvé	
6	News	News		"XX"	<GS>	Stocker	Identifiant
				"237"	<US>	Stocker	Précédant
				"238"	<US>	Stocker	Suivant
				"254"	<US>	Stocker	Série
				"255"	<US>	Stocker	Date
				"256"	<US>	Stocker	Heure
				"258"	<US>	Stocker	Texte
				"723"	<US>	Stocker	Tabulation
7	News	Top	<FS>				
8	Alarm	Alarm	(...)				
9	Alarm	Top	(...)				
10	Remove	Remove	(...)				
11	Remove	Top	(...)				

Un système de retour en arrière sur l'entrée permet également de gérer les expections de la syntaxe du flux.

3.2.2 Réassemblage des trames en dépêches

Les trames qui passent sur le flux d'informations Reuters peuvent concerner plusieurs dépêches. C'est à dire que les dépêches ne suivent pas forcément, elles peuvent se chevaucher.

3.2.2.1 Structure de données utilisée

Le réassemblage des dépêches utilise donc une structure de donnée assez évoluée dont la figure 3.3 donne une vue. Il s'agit d'un tableau de quatres listes chaînées dans

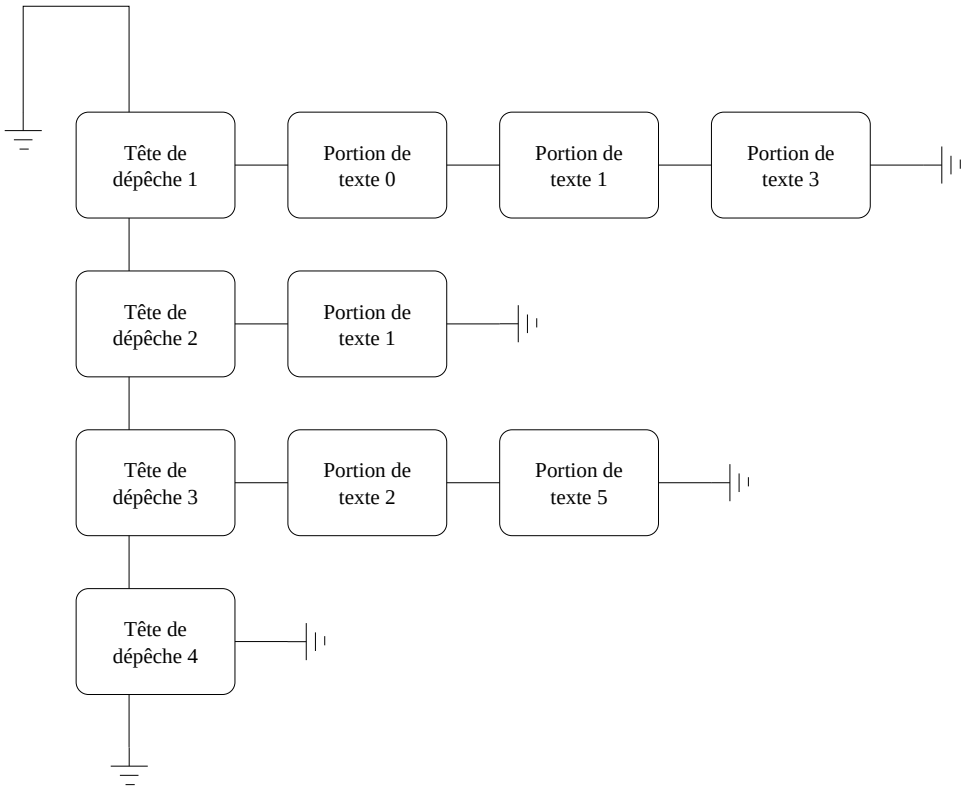

FIG. 3.3 – Rts2Db - Structure de donnée utilisée pour le réassemblage.

lesquelles les corps des textes des dépêches sont réassemblés suivant les champs : série, suivant, précédent.

3.2.2.2 Principe du réassemblage

Le réassemblage s'effectue après la détection de trame et utilise le petit algorithme suivant :

1. Si la trame est un "*remove*", alors supprimer dans la structure, la liste chaînée et la tête correspondant à cette dépêche,
2. Si la trame est une "*alarm news*", alors mettre à jour la première tête vide, ou la plus vieille tête vide, après l'avoir libérée,
3. Si la trame est une "*news*", alors :

- trouver la tête correspondant à cette dépêche, la créer en libérant une de celle-ci, si nécessaire,
- insérer dans la liste chaînée correspondante cette portion de texte, en respectant le chaînage, et en vérifiant qu'elle n'est pas déjà présente dans la liste,
- si le chaînage est complet, selectionner la dépêche si elle est intéressante, puis effacer cette dépêche dans la structure.

L'étape suivante n'est donc effectuée que lorsque qu'une dépêche est complète.

3.2.3 Sélection des dépêches

Une grande quantité de dépêches arrive par le flux d'informations **Reuters**. Mais même spécialisée dans les informations financières l'agence de presse **Reuters** diffuse aussi des dépêches à caractères beaucoup plus général.

Ainsi on trouve dans ce flux, des dépêches traitant des résultats sportifs, de politique, d'information générale...

Or, seules les informations financières concernant la **Bourse de Paris** sont nécessaires à l'approvisionement du site **YouplaBourse**.

C'est par la détection de certains codes dans les "*Product Code*" et "*Topic Code*" que l'on peut retrouver les centres d'intérêts des dépêches envoyées. **Reuters** envoie une traduction des ces codes.

Exemple :

Code	Traduction
FA	Information financière
FR	France
FB	Information générale
LFR	Langue française
SPOR	Sport
POL	Politique

Par sélection et élimination parmi ces codes, le programme détermine si une dépêche doit être sélectionnée. Si une dépêche est selectionnée, le traitement des codes et la mise en base de données est effectuée. Sinon, la dépêche est ignorée.

3.2.4 Traitement des codes particuliers

Les dépêches **Reuters** qui sont reçues, arrivent sous forme de texte **ASCII** brut, mis en forme pour une police à chasse fixe dans le cas d'un tableau. Le traitement des codes particuliers permet la mise au format **html** de certains codes pour faciliter la présentation finale dans les pages servies aux clients.

3.2.4.1 Formatage html

Il consiste au remplacement de caratères spéciaux en séquences **html**. Ainsi, les caractères accentués, la ponctuation, et tous les autres symboles qui le nécessites, sont remplacés par leurs séquences **html** correspondantes.

De plus, s'il s'agit d'un tableau, des balises pour obtenir une police particulière à chasse fixe sont insérées pour encapsuler le texte, et l'on vérifie que les séries d'espaces seront bien respectées.

Enfin, s'il s'agit de la première ligne du texte, des espaces sont insérés avant pour améliorer la lisibilité et de même pour toutes les premières lignes de paragraphes dans un texte *normal*.

Exemple :

Le texte de la dépêche envoyée en tableau :

```
PARIS, 31 août (Reuters) - Les Bourses en Europe jeudi à 8h55 GMT.
```

Deviens :

```
<FONT SIZE="3"><B><PRE>    PARIS, 31 ao&ucirc;t (Reuters) - Les Bo
urses en Europe jeudi &agrave; 8h55 GMT.</PRE></B></FONT>
```

Et le texte de la dépêche normale suivant :

```
BERLIN, 31 août (Reuters) - Malgré la hausse des prix pétroliers,
la zone euro ne présente pas de dangers d'inflation, a déclaré je
udi dans une interview à Reuters Heribert Zitzelsberger, vice-min
istre allemand des Finances.
"Nous n'avons actuellement pas d'inquiétude concernant l'inflati
on, malgré le niveau élevé des prix pétroliers, et je suis convai
ncu que l'euro va progressivement se stabiliser", a-t-il ajouté
à quelques heures d'une décision de la Banque centrale européenne
 sur un probable relèvement des taux d'intérêt.
```

Devient :

```
    BERLIN, 31 ao&ucirc;t (Reuters) - Malgr
&eacute; la hausse des prix p&eacute;troliers, la zone euro ne pr
&eacute;sente pas de dangers d&acute;inflation, a d&eacute;clar
&eacute; jeudi dans une interview &agrave; Reuters Heribert
Zitzelsberger, vice-ministre allemand des Finances.<BR>
    &quot;Nous n&acute;avons actuellement pas
d&acute;inqui&eacute;tude concernant l&acute;inflation, malgr
&eacute; le niveau &eacute;lev&eacute; des prix p&eacute;troliers,
et je suis convaincu que l&acute;euro va progressivement se stabil
iser&quot;, a-t-il ajout&eacute; &agrave; quelques heures d&acute;
une d&eacute;cision de la Banque centrale europ&eacute;enne sur un
probable rel&egrave;vement des taux d&acute;int&eacute;r&ecirc;t.
<BR>
```

3.2.4.2 Remplacement des références aux valeurs

L'agence de presse **Reuters** agrémente ses dépêches de références aux valeurs concernées par leurs dépêches.

Ces références apparaissent, dans le texte brut, sous la forme suivante :

```
<mnémonique.place>
```

Ainsi, un texte brut faisant référence à "France Télécom" à la forme suivante :

```
France Télécom <FTE.PA> maintient
```

Le remplacement de ce code s'effectue de la manière suivante :

1. Recherche dans une table dans la base de données, d'une correspondance code **Reuters** / code **Sicovam**[5] ,
2. Si le code **Sicovam** est trouvé[6] , et si le texte n'est pas un tableau (pour ne pas casser la mise en page) le code original est remplacé par un lien vers une page du site **YouplaBourse** affichant la cotation de la valeur en question,
3. Si le code **Sicovam** n'est pas trouvé, et si le texte n'est pas un tableau, le code original est simplement supprimé,
4. Pour un tableau, que le code **Sicovam** soit trouvé ou qu'il ne le soit pas, le code original Reuters est remplacé par un même nombre d'espaces, pour conserver la mise en page.
5. S'il est trouvé, le code **Sicovam** est conservé dans une liste qui sera ultérieurement stockée dans la base de données avec l'identifiant de la dépêche correspondante.

Ansi, le texte brut montré plus haut en exemple, deviendrait, dans le cas d'un texte normal :

```
France T&eacute;l&eacute;com <A HREF="#" onClick="self.opener.
location='cotation.phtml?sico=13330';">[13330]</A> maintient
```

Et dans le cas d'un tableau, le code serait remplacé par une série de 8 espaces :

```
France T&eacute;l&eacute;com          maintient
```

Le traitement des dépêches se termine par la mise en base de données de ces dernières.

3.2.5 Mise en base de données des données réassemblées

La mise en base de données des dépêches se fait au travers l'interface **MySQL** pour les programmes en **langage C**.

L'insertion s'effectue en cinq phases, dans quatres tables différentes de la manière suivante :

1. Insertion de la dépêche dans la table des dépêches, on conserve dans cette table : la date, l'heure, l'identifiant **Reuters**, la chaîne des "*Topic Codes*", la chaîne des "*Product Codes*", l'indicateur de tableau, le titre formaté et le texte formaté,
2. Récupération de l'identifiant de cette dépêche dans la table,
3. Insertion de chaque "*Topic Codes*" dans un table dédiée, avec l'identifiant de la dépêche correspondante,
4. Insertion de chaque "*Product Codes*" dans un table dédiée, avec l'identifiant de la dépêche correspondante,
5. Insertion de chaque code **Sicovam** trouvé, dans un table dédiée, avec l'identifiant de la dépêche correspondante.

Cette organisation des données permet des retrouver rapidement les dépêches concernant un "*Topic Codes*" ou un "*Product Codes*" particulier, ou encore une valeur particulère, à l'aide de son code **Sicovam**.

3.3 Fin du traitement

Tout le traitement effectué par '***rts2db***' est décrit dans ce document s'effectue à Paris. Les dépêches nouvellement décodées sont ensuite envoyées sur le site d'Angers via une connection **ftp**[7] après extraction dans la base locale à Paris.

Les identifiants des dépêches dans la base sont des entiers 'auto incrémentable'. L'opération se déroule au travers un script exécuté toutes les minutes à l'aide du service **cron**[8], qui stocke dans la base, l'identifiant de la dernière dépêche envoyée :

- si l'identifiant de la dernière dépêche traitée est plus petit que le plus grand identifiant présent dans la base alors :
- extraire la ou les dépêches dont l'identifiant est supérieur au dernier identifiant stocké,

[5] Le code **Sicovam**, code référençant de manière unique, une valeur à la **Bourse de Paris**, étant aussi celui utilisé pour référencer les valeurs sur le site **YouplaBourse**.

[6] Il est possible que le code **Sicovam** ne soit pas trouvé, dans le cas où la valeur en question n'est pas côtée à Paris, mais sur une autre place. Ce serai par exemple le cas pour "`<TVDG.DE>`" qui référence une valeur en Allemagne.

[7] **ftp** : file transfert protocole.

[8] La '*man page*' de cron dit, sans préciser quelle est l'origine du nom du démon :
cron : daemon to execute scheduled commands (Vixie Cron).

- les mettre dans un format de fichier défini, en un seul fichier,
- les envoyer par **ftp** sur le site d'Angers.

De cette manière, les dépêches sont conservées dans leur intégralité à Paris, ce qui permet une récupération de celles-ci en cas de besoin.

Chapitre 4

La réplication avec MySQL

4.1 Pourquoi la réplication ?

La société **Unimédia Communication** reçoit en direct par satellite, les cotations des différentes valeurs de la bourse de Paris, et de quelques autres places boursières en Europe. Toutes ces informations sont décodées et traitées par un programme puis stockées dans une unique base de données. Cette base de données sert de support informatif à différents clients (au sens informatique et commercial), sous la forme de différents sites internet.

Devant le succès des ces différents sites, et l'augmentation de l'exploitation de cette base de données, le besoin de trouver une solution pour *copier* de manière transparente ces informations, tout en les laissant accessibles d'une manière identique, s'est fait fortement sentir.

La **réplication** de la base de données est la solution qui a été envisagée.

Ce document constitue la description de la mise en oeuvre de cette solution avec **MySQL**.

4.2 Description de l'existant

L'existant se décompose en deux parties :
- la partie matérielle,
- la partie logicielle.

4.2.1 L'existant matériel

Unimédia Communication est équipé en interne d'un réseau **Ethernet** en 100Mb/s, et d'une *batterie* de machines allant du "simple" *Pentium* à *200 MHz* jusqu'à des machines beaucoup plus puissantes comme des *Bi-Pentium III* à *650 MHz* équipées de *1Go* de mémoire.

Une seule de ces dernières machines est dédiée à la base de données à répliquer.

4.2.2 L'existant logiciel

Le système de traitement de l'information, depuis le satellite jusqu'à la base de données, a été construit autour de machines **Linux** et du SBGDR[1] **MySQL**.

Tous les programmes de traitement de l'information, développés en interne en **langage C**, sont dédiés à ce système **Unix**, et au SGBDR **MySQL** dans sa version

[1] SGBDR : Système de Gestion de Base de Données Relationnel.

3.22.32. Par conséquent, changer de SGBDR demanderait un travail considérable de réadaptation des programmes écrits pour traiter le flux d'information.

4.3 Solution apportée

MySQL, dans la version utilisée jusque là, ne supporte malheureusement pas la réplication. Cette fonctionnalité n'apparaît qu'a partir de la version **3.23.15** et plus. Mais, dans un souci de compatibilité avec l'existant, il n'est pas envisageable de changer de système de base de données.

Par conséquent, c'est une mise à jour de **MySQL** qui va pouvoir nous permettre de mettre en place la réplication de la base de données.

Cette documentation décrit la mise en oeuvre de la réplication d'une base de données **MySQL** dans sa version `3.23.18-alpha-log`, à partir du fichier "`mysql-3.23.18-alpha.tar.gz`" contenant les fichiers sources du système de base de données.

4.3.1 Mise en oeuvre du SGBDR

La mise en oeuvre décrite ici a été testée avec succès sur trois systèmes légèrement différents : un **Linux Mandrake 6** (`Venus`) avec un noyau `2.2.10`, un **Linux RedHat 6.2** (`Zoot`), avec un noyau `2.2.14`, et enfin un **Linux RedHat 6.1** (`Cartman`), avec un noyau `2.2.12`. Les deux premières machines sont identiques, il s'agit de *Pentium III* à *550MHz* avec *128Mo* de mémoire, la dernière machine est un *Pentium Pro* à *200MHz* avec *64Mo* de mémoire.

Toutes les manipulations suivantes doivent être effectuées en tant que *super-utilisateur* (i.e. : `root`) des machines.

4.3.1.1 Décompression des fichiers sources

Le répertoire d'accueil des fichiers sources de **MySQL** est : "`/usr/local/`". La première manipulation consiste à copier le fichier "`mysql-3.23.18-alpha.tar.gz`" dans ce répertoire. Il faut ensuite le décompresser.

Comme la documentation l'indique, mais seulement lorsque le fichier est décompressé, décompresser une archive "`.tar.gz`" se fait à l'aide des utilitaires **Unix** "`tar`" et "`gzip`".

Enfin, créer un lien symbolique du répertoire "`mysql`" vers "`mysql-3.23.18-alpha`" permettra d'éviter les inconvénients d'un nom de répertoire aussi long.

Ceci peut se résumer par ces quelques lignes sur un *shell* (en tant que `root` donc) :

```
shell> cd /usr/local/
shell> mv [where]/mysql-3.23.18-alpha.tar.gz .
shell> tar xvzf mysql-3.23.18-alpha.tar.gz
shell> ln -s mysql-3.23.18-alpha mysql
```

4.3.1.2 Compilation de la base de données

La compilation passe par deux phases : la configuration, puis la compilation proprement dite.

La commande "`configure`" fait un inventaire automatique des bibliothèques et utilitaires disponibles sur la machine, et crée les fichiers "`Makefile`" adaptés à la compilation.

"`configure`" accepte un grand nombre de paramètres. Il sont décrits dans la documentation, mais nous laisserons les paramètres par défaut.

La commande "`make`" lance ensuite la compilation avec les fichiers "`Makefile`" générés par la commande précédente.

Sur le *shell*, depuis le point précédent, ceci donne :

```
shell> cd mysql
shell> ./configure
shell> make
```

4.3.2 Installation

La phase d'installation achève la compilation.

4.3.2.1 Installation des fichiers

La commande "`make install`" permet d'installer les fichiers compilés précédemment. C'est à dire, pour le principal, de copier les nouvelles bibliothèques et les nouveaux fichiers exécutables dans les répertoires appropriés et accessibles dans les chemins de recherche du système. Cette commande ne modifie cependant pas les scripts de démarrage qui doivent être modifiés manuellement pour lancer le démon après chaque réinitialisation du système.

Pour achever complètement l'installation, il faut ensuite installer les bases de données par défaut, **MySQL** gérant un grand nombre de ses propres paramètres (liste d'utilisateurs, de tables...) dans la base de données nommée par défaut "`mysql`". Le script "`mysql_install_db`" effectue cette tâche.

Sur le *shell*, depuis le point précédent, ceci donne :

```
shell> make install
shell> mysql_install_db
```

4.3.2.2 Lancement du démon `mysqld`

Comme dans les versions précédentes de **MySQL**, "`mysqld`" se lance grâce au script "`safe_mysql`". Mais contrairement aux versions précédentes, "`safe_mysql`" nécessite désormais l'option "`--user root`" pour pouvoir lancer le démon en tant que super-utilisateur.

Sur le *shell*, depuis le point précédent, ceci donne :

```
shell> safe_mysqld --user root &
```

4.3.3 Mise en oeuvre de la réplication

4.3.3.1 Création d'un utilisateur dédié à la réplication

La manoeuvre consiste à ajouter à la liste des utilisateurs de la base de données ***maître***, un utilisateur particulier qui sera utilisé pour la réplication.

J'utilise dans mon exemple, un utilisateur nommé "`replic`". Il doit avoir le droit de se connecter depuis tous les ***esclaves***, avec tous les droits nécessaires à la consultation des bases à répliquer.

Il est possible de créer des utilisateurs **MySQL** par plusieurs moyens, je propose ci-dessous, un script utilisable depuis la ligne de commande et qui effectue cette tâche : "`adduser.my`"

```
#!/bin/sh
/usr/local/mysql/bin/mysql -u root mysql << EOF
insert into user values("$1","$2","$3","Y","Y",
"Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y","Y");
```

```
EOF
/usr/local/mysql/bin/mysqladmin -u root reload
```

Ce script s'appelle avec 3 paramètres :
- La machine distante ou locale depuis laquelle l'utilisateur créé sera autorisé à se connecter,
- Le nom de cet utilisateur,
- Le mot de passe encrypté de cet utilisateur.

Donc, sur le ***maître***, depuis le point précédent de la ligne de commande, l'ajout d'un utilisateur "`replic`" pouvant se connecter depuis un ***esclave*** nommé "`esclave.unimedia.fr`" avec un mot de passe encrypté "`bidon`" se fait de la manière suivante :

```
shell[maître]> ./adduser.my "esclave.unimedia.fr" "replic" "bidon"
```

Remarque :

Le dernier paramètre peut être omis. L'utilisateur peut alors se connecter sans mot de passe.

4.3.3.2 Copie des bases de données initiales

Avant même de commencer à configurer les ***esclaves***, et après avoir chargé les bases de données sur le ***maître*** sans l'avoir configuré pour la réplication, il faut récupérer des "`dump`" (voir la commande "`mysqldump`") des bases de données présentes sur le ***maître***.

Le but est de copier des bases de données identiques au départ, sur les ***esclaves*** et le ***maître***.

Par exemple, copier la base "`test`" du ***maître*** vers l'***esclave*** peut se faire de la manière suivante :

```
shell[maître]> mysqldump test > test.dump
[...] Faire le transfert de fichier entre les deux machine,
      par FTP par exemple.
shell[esclave]> mysqladmin drop test
    --> pour détruire une base de données 'test' sur la
        machine esclave
    --> si elle existait déjà.
shell[esclave]> mysqladmin create test
shell[esclave]> mysql test < test.dump
```

4.3.3.3 Fichiers de configuration

La phase de configuration du ***maître*** comme des ***esclaves*** doit se faire après avoir arrêté le démon "`mysqld`". Dans les deux cas, il s'agit d'éditer (ou de créer si nécessaire) un fichier "`/etc/my.cnf`".

Pour le *maître* :

La configuration du ***maître*** se résume à l'ajout des lignes suivantes dans le fichier de configuration :

```
# TEST 'my.cnf' file for a master.
[mysqld]
log-bin
```

Pour l'*esclave* :

La configuration revient à ajouter les lignes renseignant le ***maître*** :

```
# TEST 'my.cnf' file for a slave
[mysqld]
master-host=<hostname or IP of the master>
master-user=<replication user name>
# dans notre cas : replic
master-password=<replication user password>
```

Il suffit ensuite, normalement, de redémarrer les démons "`mysqld`" sur le ***maître*** et sur les ***esclaves***, les deux bases sont alors synchronisées.

4.4 Fonctionnalités testées

La conclusion de ce document s'organise autour d'un inventaire des fonctionnalités testées avec succès.

4.4.1 Fonctionnalité SQL

Après une installation complète sur les deux premières machines, le système exécutait correctement les opérations synchronisées suivantes de l'***esclave*** sur le ***maître***, c'est à dire répercution des opérations effectuées sur le ***maître*** vers l'***esclave*** :

- création de nouvelles base de données,
- création de nouvelles tables,
- insertion d'éléments dans les tables,
- modifications ou "`updates`" d'éléments dans les tables,
- suppressions d'éléments dans les tables,
- suppressions de tables,
- suppressions de bases de données.

4.4.2 Résistance à l'arrêt des services

Une certaine forme de résistance à la panne a elle aussi été testée avec succès, le cas de chute correcte des différents services :

- reprise correcte de la synchronisation après arrêt correct et redémarrage de l'***esclave***,
- reprise correcte de la synchronisation après arrêt correct et redémarrage du ***maître***.

La résistance à la panne *matérielle* (i.e. : *crash* du système sur une panne de courant ou un défaut matériel) n'a pas été testée. La documentation précise que la reprise sur ce type de panne peut être correcte si les processus constituant le service ont eu le temps, avant le *crash* du système, de vider complètement leur mémoire cachée vers les fichiers constituant la base de données[2].

4.4.3 Insertion d'un nouvel *esclave*

Enfin, l'ajout d'un nouvel ***esclave*** sur un système en marche a été testé avec succès de la manière suivante, avec la dernière machine :

- préparation d'un nouvel ***esclave*** à partir des bases de données "*pré-synchronisation*",
- déclaration du nouvel ***esclave*** auprès du ***maître***,

[2] En anglais dans la documentation : "*[...] Unclean shutdowns might produce problems, especially if disk cache was not synced before the system died. [...]*"

- configuration de l'***esclave***,
- insertion de l'***esclave*** dans le système.

A partir de ce dernier point, le nouvel ***esclave*** se synchronise avec le ***maître***, pour atteindre le même état que ce dernier, et assimile donc toutes les opérations effectuées depuis la mise en synchronisation du système.

4.4.4 Ce qui ne marche pas

Il n'y a qu'une fonctionnalité que l'on aurait pu espérer obtenir par la réplication, mais que la version testée de **MySQL** n'implémente pas, c'est la répercussion des opérations **SQL** effectuées sur un ***esclave***, vers le ***maître***, puis sur tous les ***esclaves***.

4.5 Correction

4.5.1 Version de MySQL

La version de **MySQL** à laquelle il est fait référence dans cette partie du document, c'est à dire la version `3.23.18-alpha-log`, est partiellement "bugée". C'est du moins la conclusion à laquelle nous sommes parvenus après expérimentation plus poussée de cette version.

En effet, la version `3.23.18-alpha-log` de **MySQL** exécutait en près de 1 heure 30 minutes, un programme qui était terminé au bout de 5 minutes maximum sur la machine de production, avec la version `3.22.32` de **MySQL**.

Malgrès l'aide et le soutien de l'équipe de développement de **MySQL**, nous n'avons réussi à tirer qu'une seule conclusion : la version `3.23.22-beta-log` de **MySQL** ne présente plus ce disfonctionnement.

Par conséquent, c'est cette dernière version, la version `3.23.22-beta-log` de **MySQL**, que nous utiliserons.

Chapitre 5

Yalta

Yalta : Yet Another Language of Technical Analysis.

5.1 Introduction

Le but de ce projet est la création d'un langage de programmation et d'un logiciel exécutant ce langage, pour permettre la manipulation de vecteurs de cotations, nécessaire à l'analyse technique boursière.

Ces opérations doivent constituer pour l'utilisateur final, une aide à la décision boursière.

Cette application est écrite en langage **C**, et les fonctionnalités de compilation sont développées à l'aide de **Yacc**.

Les cotations sont stockées dans une base de données distante.

L'interpréteur réalisé doit, à terme, être appelé par une "sur-couche" écrite en **Java**, permettant la saisie des indicateurs et la présentation des résultats sous formes de graphiques.

5.2 Données du problème

Les données régissant le problème sont les suivantes.

5.2.1 Vecteurs de cotations

Les vecteurs de cotations, stockent pour une valeur boursière, par jour ouvrable (*date*), les cotations à l'ouverture (*open*), à la fermeture (*close*), la plus haute (*high*) et la plus basse (*low*) de la séance, ainsi que le volume de transactions (*volume*).

Les vecteurs de cotations sont donc une suite de ces informations élémentaires.

Descripteur	*date*	*open*	*high*	*low*	*close*	*volume*
Entrée	d	o	h	l	c	v
Type	entier	réel	réel	réel	réel	entier

Les vecteurs peuvent êtres assimilés à des tableaux de ces éléments, auquels il faut inclure :

- Le nombre de cotations mémorisées pour cette valeur, c'est à dire la taille des tableaux,
- Un tableau de dates correspondantes.

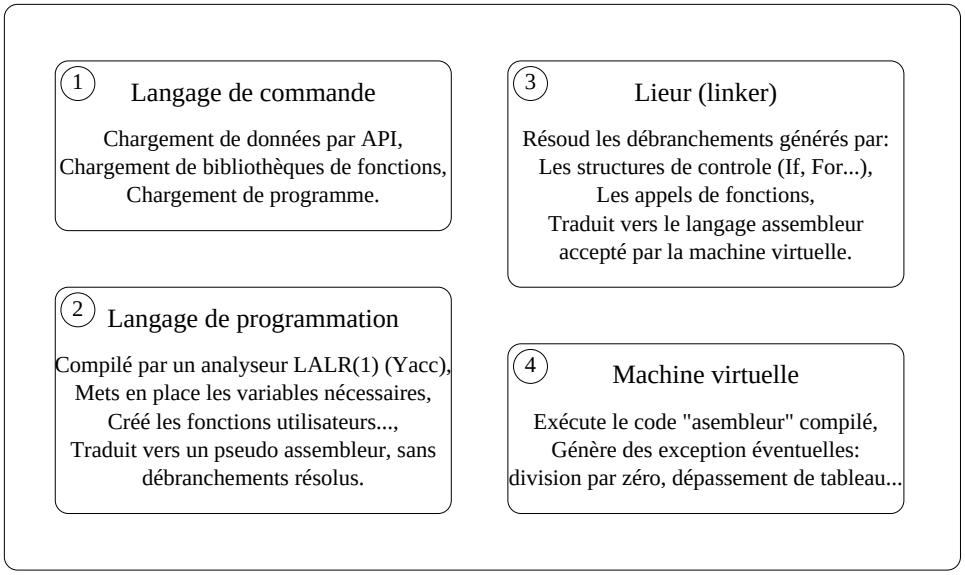

FIG. 5.1 – Yalta - Forme générale du compilateur.

5.2.2 Stockage distant

Ces cotations sont stockées dans une base de donnée distante[1].

Deux possibilités semblent envisageables :

- L'insertion d'une couche API permettant à l'interpréteur d'interroger directement la base de données,
- L'insertion de moyen pour créer, depuis l'entrée standart, des vecteurs de ce type, et laisser à la sur-couche **Java**, le soin d'interroger la base de données.

Quelque soit la méthode adoptée, l'interpréteur devra donc disposer d'un *méta-langage* permettant le chargement de vecteurs.

5.2.3 Résultats

Les résultats des opérations proposées sont des indicateurs réels (par exemple : la cotation moyenne à la fermeture), des vecteurs de réels (par exemple : vecteurs de cotations moyennes sur 10 jours à la fermeture), des cotations complètes...

Dès qu'il s'agit de stocker des vecteurs, il semble nécessaire de stocker parallèlement une date.

5.3 Présentation générale de l'application

L'application que nous développons doit être appelée par la sur-couche écrite en **Java**. Le mode de communication entre la couche **Java** et celle-ci, reste à définir. Il pourra s'agir d'un système de "tube" (au sens **Unix** du terme), ou bien éventuellement, d'un petit protocole de communication traité à l'aide de **TPC/IP**.

Quelque soit le mode de communication, plusieurs sous parties composent cette application (c.f. : la figure 5.1).

[1] En l'occurence, par un SGDB **MySQL**, sur un serveur **Linux**.

5.3.1 Choix du langage de programmation

Plusieurs solutions ont étés envisagées pour mettre à la disposition de l'utilisateur final, un langage de programmation, à la fois suffisamment puissant pour ce type d'application, et suffisamment abordable pour un utilisateur pas forcement initié à la programmation. Plusieurs solutions ont paru plus ou moins acceptables. Elle sont les suivantes.

5.3.1.1 Langage impératif procedural

La possibilité d'implémenter un compilateur pour un langage de programmation du type impératif a été envisagé. Un tel type de langage peut très rapidement, reprendre les caractéristiques de langages de programmation connus (la grammaire, les structures de contrôles...) tels que le langage **C** ou le langage **Pascal**.

C'est la cas, par exemple, de "*Easy Langage*"[2], le langage dédié aux calculs d'indicateurs accompagnant le logiciel *TradeStation 2000i*, de **Omega Research**. Ce langage, explicitement typé, semble fortement s'inspirer de la syntaxe du langage **Pascal**, avec le même type de structures de contrôles (par exemple les blocs d'instructions se présentant sous la forme "`Begin ... end ;`").

Malheureusement, il semble que son utilisation, malgré la puissance qu'il met à la disposition des utilisateurs, ne soit pas forcément aisée pour un novice n'ayant, à priori, aucunes notions de programmation : pour preuve, le nombre de professionnels se mettant à la disposition de ces utilisateurs sur l'Internet, pour écrire leurs programmes et leurs indicateurs.

Une solution similaire a donc été écartée.

5.3.1.2 Extension du langage Java

Pour des raisons de simplifications de la phase de développement du produit, et d'homogénéité de celui-ci, la possibilité d'utiliser le langage **Java** comme langage de calculs d'indicateurs a été envisagée.

En effet, **Java**, langage de programmation orienté objet, permet le chargement dynamique de nouvelles classes d'objets. La solution envisagée se décompose donc de la manière suivante :

- Mise à disposition de l'utilisateur, d'une bibliothèque de classes d'objets comprenant une classe "Cotation" permettant la manipulation des informations désirées,
- Mise en place d'un outil d'édition permettant l'extension de cette classe, pour y inclure les méthodes supplémentaires correspondant au besoin de l'utilisateur : le calcul d'indicateurs,
- Compilation et chargement de la classe ainsi étendue par l'utilisateur.

De toute évidence, une telle solution est rapide à mettre en oeuvre. De plus, le langage de programmation mis ainsi à la disposition de l'utilisateur, n'est ni plus, ni moins que du "**Java**", spécialisé pour le problème donné, offrant toute la puissance associée à un tel langage, orienté "objet".

Les inconvénients à une telle solution ne manquent pourtant pas. En effet, s'il parait surréaliste de faire programmer à un novice dans un langage proche du **Pascal**, c'est encore plus le cas avec un langage comme **Java**, même si la documentation et les ouvrages sur le sujet ne manquent pas. De plus, une telle solution nécessite l'installation d'un compilateur **Java** complet sur le poste client de l'usager, ce qui n'est pas, envisageable.

[2] *Easy Langage* ou "*ela*".

5.3.1.3 Langage fonctionnel

Plusieurs langages existants, dédiés au calcul (statistique, matriciel...) , sont des langages de programmation fonctionnels. C'est le cas, par exemple, de **SPlus** de MathSoft, ou encore de **Pari** développé par le laboratoire A2X du C.N.R.S. de Bordeaux. Malheureusement, ces langages sont très évolués, et non spécialisés. De plus, il est assez difficile d'évaluer leur portabilité, et donc d'envisager de les utiliser pour notre application.

Un logiciel dédié à l'analyse technique, **MetaStock** de *Equis*, supporte une programmation simple d'indicateur, à l'aide d'un langage, lui aussi fonctionnel.

Il manipule et calcule des vecteurs de réels, issus des cotations. Il est spécialisé pour ce type de calculs. Le formalisme qu'il utilise n'est pas forcément proche des formalismes habituels, issus des mathématiques, utilisés pour les calculs sur les vecteurs, mais il est plus adapté.

D'un autre côté, certaines formes, pourtant présentes dans d'autres langages, ne sont pas forcément très adaptées, et plusieurs améliorations pourraient être effectuées.

Exemple :

Référencer les cotations du jour précédent se fait avec le langage **MetaStock**, de la manière suivante (cotations à la fermeture il y a un jour) :

```
ref ( c , -1 )
```

Alors qu'une symbolique reprenant celles des vecteurs, pourrait permettre quelque chose du genre :

```
c [ 1 ]
```
[3]

Le langage **Metastock** semble malgré tout être une base intéressante de travail, pour construire notre propre langage.

5.3.2 Description de *Yalta*

Cette partie décrit la liste des fonctions qu'il a été prévu d'implémenter dans le langage **Yalta**. Certaines de ces fonctions sont souvent inspirées de ce que le langage de **MetaStock** permet de faire, et si d'ailleurs, certaines fonctions présentes dans l'index qui suit, ne sont pas implémentées, c'est simplement parce que cet index est construit sur celui des fonctions proposées dans **MetaStock** lui-même.

Le projet final doit permettre de pouvoir faire tout ce qui est déjà possible avec **MetaStock**, et plus encore.

Voici donc les types, les fonctions et les constantes telles qu'elles ont été implémentées dans le langage **Yalta**.

5.3.2.1 Types

Les types utilisés dans cet index de fonctions, sont les suivants :
- vector ⇔ un vecteur de réel, d'entier (ramené à un réel) ou de booléen (0 pour *faux*, n'importe quelle autre valeur sinon),
- int ⇔ un entier (une période, une date...),
- real ⇔ un réel (généralement, une cotation),
- boolean ⇔ un vecteur de booléens (0 pour *faux*...)
- const ⇔ les constantes du type "SIMPLE", "TRIANGULAR", "EXPONENTIAL" (...) et leurs abréviations respectives,

[3] Cette symbolique, non conforme avec celle des mathématiques courante pour les vecteurs, est inspirée de ce que le langage *ela* de **Omega Research** permet de faire.

- string $\Leftrightarrow$ les chaînes de caractères, type supporté dans certaines opérations avec la couche graphique, mais pas par le langage **Yalta** lui-même.

Les vecteurs évalués par **Yalta**, sont toujours des vecteurs de réels. Cependant, de par le sens de l'opération effectuée, le type peut être "*réduit*". Le type "*maximum*" de la fonction est donné dans les tableaux suivants, en colonne "**Résultat**".

5.3.2.2 Fonctions standart

Les tableaux suivants proposent donc une méthode envisageable pour implémenter dans notre langage, des fonctions :

- * $\Rightarrow$ câblée dans la machine virtuelle.
- G $\Rightarrow$ dans la couche graphique.
- S $\Rightarrow$ câblée en soft (définition de fonction dans ce langage le langage **Yalta** lui même).

La dernière colonne "**Y**" , si elle est cochée '*', indique que la fonction est réellement implémentée dans le langage **Yalta**, avec le type décris. Si cette colonne est cochée 'X', c'est que la fonction est partiellement implémentée par le langage[4].

Symboles :

Description	Syntaxe	Résultat	Câblée	Y
Negative values	$-$ vector	real	*	*
Multiplication	vector $*$ vector	real	*	*
Division	vector /vector	real	*	*
Addition	vector + vector	real	*	*
Substraction	vector $-$ vector	real	*	*
Less than	vector $<$ vector	boolean	*	*
Greater than	vector $>$ vector	boolean	*	*
Less than or equal to	vector $<=$ vector	boolean	*	*
Greater than or equal to	vector $>=$ vector	boolean	*	*
Equal to	vector $==$ vector	boolean	*	*
Not equal to	vector $!=$ vector	boolean	*	*
Logical "And"	vector & vector	boolean	*	*
Logical "Or"	vector | vector	boolean	*	*
Variable assignment	name = vector		*	*
Incrementation	int++	int	*	*
Plotted indicator	P	real	G	
Previous value	$Prev$	real	*	

La fonction "*Variable assignment*" fait apparaître un type "*name*". Il s'agit en fait d'un identifiant pour un nom de variable.

[4]Les fonctions partiellement implémentée par le langage sont généralement, des fonction n'acceptant pas certaines constantes en paramètre.

A :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Absolute Value	abs(vector)		real	*	
Accumulation/Distribution	ad()		real	S	*
Accumulation Swing Index	aswing(real)				
Addition	add(vector , vector)	+	real	*	*
Alert	alert(boolean , int)		boolean		
Arc Tangent	atan(vector , vector)		real	*	
Aroon Down	aroondown(int)				
Aroon Up	aroonup(int)				
Average Directional Movement	adx(int)				
Average True Range	atr(int)				

B :

Description	Syntaxe	Résultat	Câblée	Y
Bars Since	barsince(vector)			
Bollinger Band Bottom	bbandbot(vector , int)	real	S	*
Bollinger Band Top	bbandtop(vector , int)	real	S	*
Buying Pressure	buyp()			

C :

Description	Syntaxe	Résultat	Câblée	Y
Ceiling	ceiling(vector)	int	*	
Chaikin A/D Oscillator	co()			
Chaikin's Money Flow	cmf(int)			
Chande Momentum Oscillator	cmo(vector , int)			
Commodity Channel Index (EQUIS)	ccie(int)			
Commodity Channel Index (Standard)	cci (int)			
Commodity Selection Index	csi(int , real , real , real)			
Correlation Analysis	correl(vector , vector , int , int)			
Cosine	cos (vector)	real	*	
Cross	cross (vector , vector)	boolean	*	
Cumulates	cum(vector)	real	*	*

D :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Day Of Month	dayofmonth()				
Day Of Week	dayofweek()		int		
Delta	delta(const , int , real , real , real)				
Dema	dema(vector , int)				
Demand Index	di()				
Detrended Price Oscillator	dpo(int)				
Directional Movement Index	dx(int)				
Directional Movement Rating	adrx(int)				
Divergence	divergence(vector , vector , real)		int	*	
Division	div(vector , vector)	/	real	*	*
Dynamic Momentum Index	dmi(vector)		real		

E,F :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Ease of Movement	emv(int , const)				
Exponent	exp(vector)	real		*	
Fast Fourier Transform	fft(vector , int , int , const , const)				
Floor	floor(vector)		int	*	
Forecast Oscillator	forecastosc(vector , int)				
Fraction	frac (vector)		real	*	

G :

Description	Syntaxe	Résultat	Câblée	Y
Gamma	gamma(const , int , real , real , real)			
Gap Down	gapdown()	boolean	*	
Gap Up	gapup()	boolean	*	

H :

Description	Syntaxe	Résultat	Câblée	Y
Herrick Payoff Index	hpi(real , real)			
Highest	highest(vector)	real	*	*
Highest Bars Ago	highestbars(vector)	real	*	
Highest High Value	hhv(vector , int)	real	*	*
Highest High Value Bars Ago	hhvbars(vector , int)	real	*	
Highest Since	highestsince(int , boolean , vector)	real	*	
Highest Since Bars Ago	highestsincebars(int , boolean, vector)	real	*	

I,K :

Description	Syntaxe	Résultat	Câblée	Y
If	if(boolean , vector , vector)	real	*	*
Inertia	inertia(int , int)			
Input	input(string , real , real , real)	real (non vector)	G	
Inside	inside()			
Integer	int(vector)	int	*	
Intraday Momentum Index	imi(int)			
Klinger Volume Oscillator	kvo()			

L :

Description	Syntaxe	Résultat	Câblée	Y
Last Value in Data Array	lastvalue(vector)	real	*	
Linear Regression Indicator	linearreg(vector , int)			
Linear Regression Slope	linearregslope(vector , int)			
Logarithm (natural)	log(vector)	real	*	
Lowest	lowest(vector)	real	*	*
Lowest Bars Ago	lowestbars(vector)	real	*	
Lowest Low Value	llv(vector , int)	real	*	*
Lowest Low Value Bars Ago	llvbars(vector , int)	real	*	
Lowest Since	lowestsince(int , boolean , vector)	real	*	
Lowest Since Bars Ago	lowestsincebars(int , boolean , vector)	real	*	

M :

Description	Syntaxe	Alias	Résultat	Câblée	Y
MACD standart divergence	macd_std()		real	S	*
MACD standart trigger	macd_std_trigger()		real	S	*
MACD standart indicator	macd_std_indicator()		real	S	*
MACD divergence	macd(int,int,int)		real	S	*
MACD trigger	macd_trigger(int,int,int)		real	S	*
MACD indicator	macd_indicator(int,int,int)		real	S	*
Market Facilitation Index	marketfacindex()				
Mass Index	mass(int)				
Maximum	max(vector , vector)		real	*	
Median Price	mp()		real	S	*
MESA Lead Sine	mesaleadsine(int)				
MESA Sine Wave	mesasine(int)				
Midpoint	mid(vector , int)		real	S	*
Minimum	min(vector , vector)		real	*	
Minus Directional Movement	mdi(int)				
Modulus	mod(vector , vector)		int	*	
Momentum	mo(int)		real	*	*
Money Flow Index	mfi(int)				
Month	month()				
Moving Average	mov(vector , int , const)		real	*	X
Multiplication	mul(vector , vector)	*	real	*	*

N,O :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Negative	neg(vector)	-	real	*	*
Negative Volume Index	nvi()				
On Balance Volume	obv()				
Option Expiration	nextoptionexp()				
Option Life	life(int)		int	*	
Outside	outside()		boolean	*	

P :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Parabolic SAR	sar(real , real)				
Peak Bars Ago	peakbars(int , vector , real)		int	*	
Peak Value	peak(int , vector , real)		real	*	
Performance	per()				
Plus Directional Movement	pdi(int)				
Polarized Fractal Efficiency	pfe(vector , int , int)				
Positive Volume Index	pvi()				
Power	power(vector , real)	^	real	*	
Precision	prec(vector , int)		real	*	
Price Channel High	pricechannelhigh(int)				
Price Channel Low	pricechannellow(int)				
Price Oscillator	oscp(int , int , const , const)				
Price Volume Trend	pvt()				
Projection Band Bottom	projbandbot(int)				
Projection Band Top	projbandtop(int)				
Projection Oscillator	projosc(int , int)				
Put/Call Price	option(const , int , real , real , real)				

Q,R :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Qstick	qstick(int)				
r-squared	rsquared(vector , int)				
Rally	rally()		boolean	*	
Rally With Volume	rallywithvol()		boolean	*	
Random Walk Index of Highs	rwih(int)				
Random Walk Index of Lows	rwil(int)				
Range Indicator	rangeindicator(int , int)				
Rate of Change Price	roc_price(int)			S	*
Rate of change Standart Price	roc_std_price()			S	*
Rate of change Volume	roc_volume(int)			S	*
Rate of change Standart Volume	roc_std_volume(int)			S	*
Reaction	reaction()		boolean	*	
Reaction With Volume	reactionwithvol()		boolean	*	
Reference	ref(vector , int)	v[i]	real	*	*
Relative Momentum Index	rmi(vector , int , int)				
Relative Strength Index (RSI)	rsi(vector)		int	*	
Relative Volatility Index	rvi(int)				
Round	round(vector)		int	*	

S :

Description	Syntaxe	Alias	Résultat	Câblée	Y
Selling Pressure	sellp()				
Sine	sin(vector)		real	*	
Square Root	sqrt(vector)		real	*	
Standard Deviation	stdev(vector , int)		real	*	*
Standard Error	ste(vector , int)				
Standard Error Band Bottom	stebandbot(vector , int , real)				
Standard Error Band Top	stebandtop(vector , int , real)				
Stochastic Momentum Index	stochmomentum(int , int , int)		real	S	*
Stochastic Oscillator	stoch(int , int)		real	S	*
Subtraction	sub(vector , vector)	-	real	*	*
Summation	sum(vector , int)		real	*	
Swing Index	swing(real)				

T,U :

Description	Syntaxe	Résultat	Câblée	Y
Tema	tema(vector , int)			
Theta	theta(const , int , real , real , real)			
Time Series Forecast	tsf(vector , int)			
Trade Volume Index	tvi(real)			
TRIX	trix(int)			
Trough Bars Ago	troughbars(int , vector , real)			
Trough Value	trough(int , vector , real)			
Typical Price	tipical()	real	S	*
Ultimate Oscillator	ult(int , int , int)			

V :

Description	Syntaxe	Résultat	Câblée	Y
Value When	valuewhen(int , boolean , vector)	real	*	
Variance	var(vector , int)	real	*	
Vega	vega(const , int , real , real , real)			
Vertical Horizontal Filter	vhf(vector , int)			
Volatility, Chaikin's	vol(int , int)			
Volatility, Option	volo()			
Volume Oscillator	oscv(int , int , const , const)			

W,Y,Z :

Description	Syntaxe	Résultat	Câblée	Y
Weighted Close	wc()			
Wilder's Smoothing	wilders(vector , int)			
Williams' %R	willr(int)		S	*
Williams' A/D	willa()			
Writeif	writeif(boolean , string , string)		G	
Writeval	writeval(vector)		G	
Year	year()			
Zig Zag	zig(vector , real , const)			

5.3.2.3 Constantes

Les constantes décrites ici, sont celles apparaissant dans les appels de fonction de **Yalta**.

Description	Alias	Valeur	Occurence
AMPLITUDE		0	fft
DETREND		1	fft
CALL	EC	2	delta, gamma, option, theta, vega
EXPONENTIAL	E	3	bbandbot, bbandtop, emv, mov, oscp, oscv
FUTURECALL	FC	4	delta, gamma, option, theta, vega
FUTUREPUT	FP	5	delta, gamma, option, theta, vega
MEAN		6	fft
PERCENT	%	7	roc, oscv, zig
POINTS	$	8	roc, oscv, zig
POWER		9	fft
PUT	EP	10	delta, gamma, option, theta, vega
SIMPLE	S	11	bbandbot, bbandtop, emv, mov, oscp, oscv
TIMESERIES	T	12	bbandbot, bbandtop, emv, mov, oscp, oscv
TRIANGULAR	TRI	13	bbandbot, bbandtop, emv, mov, oscp, oscv
VARIABLE	VAR	14	bbandbot, bbandtop, emv, mov, oscp, oscv
WEIGHTED	W	15	bbandbot, bbandtop, emv, mov, oscp, oscv

5.4 Langage développé

Le langage de programmation que je propose de mettre à la disposition de l'utilisateur est donc fonctionnel.

Il permet la manipulation de vecteur de cotation, pour lesquels les problèmes de dates sont gérés de manière transparente à l'utilisateur, par la machine virtuelle, ceci lors de l'exécution du code compilé.

Deux langages se distinguent : un langage de commande, et un langage de programmation.

5.4.1 Langage de commande

Au moment où ce document est finalisé, aucune idée sur le type de langage de commande n'est réellement arrêtée.

Les différents outils implémentés sont, pour le moment encore, appelés par un script **Unix**, qui, dans cet ordre :

- compile les fichiers sources,
- lie les fichiers sources entre eux, avec la librairie **Yalta** (pour les fonctions implémentées dans ce langage),
- effectue le chargement des données demandées par l'utilisateur (en utilisant le symbole décrivant la cotation à analyser),
- lance la machine virtuelle avec le programme compilé et les différentes données reccupérées,
- lance une application graphique permettant un traçage des résultats, avec les résultats produits par la machine virtuelle avec le programme compilé (c.f. : 5.10 page 64).

5.4.2 Langage de programmation

Ce langage de programmation prend ses inspirations dans diverses formes de programmation déjà existantes (cf : 5.3.1 page 35). Dans les grandes lignes, le langage de programmation **Yalta** est inspiré du **langage C** pour la syntaxe *déclarative*, et de différents langages de programmation fonctionnels pour la syntaxe des expressions.

5.4.2.1 Commentaires

Comme tout "*bon*" langage de programmation qui se doit, **Yalta** accepte des commentaires.

Pour simplifier la tâche de l'utilisateur, et lui éviter la situation ambiguë des commentaires imbriqués les uns dans les autres, la solution retenue n'est pas celle adoptée en **langage C** à la norme **ANSI**[5], mais plutôt celle des différent **C++** existant. Les commentaires sont donc placés entre "//" et un retour à la ligne.

Ce qui fait donc des lignes suivantes des commentaires valides :

```
// Commentaire simple.
   // Commentaire ne commençant pas en début de ligne.
a + b ; // Commentaire suivant une instruction.
```

D'une manière générérale, les espaces redondants, les tabulations et les retours à la ligne sont ignorés.

5.4.2.2 Gestion des types

Pour simplifier le problème de la gestion des différents types au yeux de l'utilisateur, le langage **Yalta** n'en accepte pas en tant que tel. Comme le fait **MetaStock**, **Yalta** *simule* les types, c'est à dire que dans tous les cas, une variable est traitée comme un vecteur de réels de la manière suivante et par équivalence comme un des types suivant :

- Un booléen $\Leftrightarrow$ un vecteur de taille 1, ou singleton, contenant 0 pour faux, n'importe quelle autre valeur pour vrai,
- Un entier $\Leftrightarrow$ un singleton contenant, sous forme réelle, l'entier à stocker,
- Un réel $\Leftrightarrow$ un singleton contenant le réel à stocker.

Par cette simplification, l'utilisateur n'est plus contraint par les problèmes de typages. Ceci représente pour lui un grand facteur de simplification, en particulier s'il n'est pas initié à la programmation, puisqu'il n'a pas, ainsi,à déclarer les types de ses variables, celles-ci étant systématiquement traitées comme des vecteurs de réels.

Par contre, ceci lui permet de faire des choses incontrôlées puisque les équivalences précédentes peuvent être ramenées à celle-ci :

Un booléen $\Leftrightarrow$ un entier $\Leftrightarrow$ un réel

Du point de vue de la machine virtuelle, cette simplification permet de n'avoir à traiter que des variables du type "vecteur de réels".

Ainsi, une variable peut-être déclarée à n'importe quel point d'une fonction, et à valeur dans toute la fonction dans laquelle elle est déclarée. Par défaut, une variable est initialisée comme un vecteur de taille 1 contenant la valeur "0".

5.4.2.3 Expressions

Dans le langage fonctionnel **Yalta**, l'élément le plus *petit*, sémantiquement parlant, est une expression. C'est ce qui peut être vu, du point de vue des langages impératifs, comme des instructions.

Les expressions sont séparées par des ' ; '. Elle peuvent être constituées d'identifiants de variables, d'opérateurs sur ces identifiants, ou encore d'appels de fonctions,

[5] C'est à dire pas : `/* de tels commentaires */`.

les opérateurs étant eux-mêmes pour la plupart, des *alias* vers des fonctions prédéfinies.

Yalta dispose d'une large gamme de fonctions et d'opérations, traitées au niveau de la machine virtuelle, comme des instructions élémentaires. Ce sont les instructions arithmétiques de calcul (c.f. : 5.5.1.1 page 48) et les instructions câblées (c.f. : 5.5.1.5 page 52). Cette section n'a pas pour but de les détailler toutes, mais simplement d'en donner quelques exemples pour donner la forme des expressions qu'il est possible d'utiliser avec ce langage.

Les expressions peuvent être vues comme des formules mathématiques, dont elles adoptent en partie le formalisme. Ainsi, les opérateurs arithmétiques élémentaires ("+" , "-" , "*" , "/" , "(" , ")" ...) bénéficient des règles de priorités standards.

L'opérateur dédié à l'attribution d'une valeur à une variable est le signe égal "=". Ainsi, l'expression suivante est correcte :

```
a = 0 ;
```

Elle ramène le vecteur nommé "a" à un singleton contenant la valeur "0".

La priorité des opérateurs et l'équivalence entre alias et fonctions, font des expressions suivantes, des expressions équivalentes :

```
1 + 2 * 3 ;
1 + ( 2 * 3 ) ;
add ( 1 , mul ( 2 , 3 ) ) ;
```

Remarque :

Les fonctions définies par l'utilisateur s'appellent de la même manière que les fonctions prédéfinies.

5.4.2.4 Structures de contrôle

Bien que les *structures de contrôle* ne soient pas courantes dans les langages de programmation purement fonctionnels, **Yalta** en implémente quelques unes. Voici la forme qu'elles prennent.

le *Si* ou `IF` :

La forme que prend le "`IF`" dans le langage **Yalta**, n'est pas celle adoptée par le **langage C**. Pour rester dans un esprit de programmation fonctionnelle, et bien que le "`IF`" soit, au final, implémenté comme un vrai débranchement conditionnel, il prend la forme fonctionnelle suivante :

```
if ( expr_test , expr_vrai , expr_faux ) ;
```

Où :

- `expr_test` est l'expression dont la valeur sera testée,
- `expr_vrai` est l'expression doit être la valeur sera renvoyée par le "`if`" si le test est vrai,
- `expr_faux` est l'expression doit être la valeur sera renvoyée par le "`if`" si le test est vrai.

L'inconvénient d'une telle implémentation est que chacun des arguments est **nécessaire**, y compris `expr_faux`, le "`IF`" devant pouvoir retourner une valeur.

La boucle *Tantque* ou `WHILE` :

Cette structure de contrôle prend la même forme dans le langage **Yalta** que dans le **langage C** :

```
while ( expr_test ) {
    // Corps de  la boucle,
    // c'est à dire suite d'expression ou de structures de contrôle...
}
```

Où "`expr_test`" est l'expression dont la valeur sera testée pour sortir de la boucle. Tant que la valeur du test est vraie, la boucle est exécutée.

A la différence de la forme de cette structure de contrôle en **langage C**, celle qu'elle prend dans le langage **Yalta** nécessite une valeur pour le test.

La boucle *Pour* ou `FOR` :

La structure de contrôle de la boucle "`FOR`" dans le langage **Yalta**, prend la même forme qu'en **langage C** :

```
for ( expr_init; expr_test; expr_inc ) {
    expr_corps; [ ... ]
    // Corps de  la boucle,
    // c'est à dire suite d'expression ou de structures de contrôle...
}
```

Le meilleur moyen de comprendre comment fonctionne effectivement la boucle "`FOR`", est de voir comment le compilateur traduit le langage **Yalta** dans son assembleur dédié (c.f. : 5.5.2.2 page 56).

En simplifiant, les actions s'enchaînent de la manière suivante :

1. `expr_init` est exécuté une et une seule fois,
2. `expr_test` est exécuté, et laissé dans la pile, si le résultat qu'il renvoie est faux, on sort de la boucle (on va en 5),
3. `expr_corps` est exécuté,
4. `expr_inc` est exécuté (après `expr_corps`), puis il y a retour au test (en 2),
5. fin de la boucle.

Contrairement à la forme de la boucle "`FOR`" en **langage C**, il n'est pas permis avec **Yalta**, d'omettre l'une des expressions formant l'*entête* de la boucle.

5.4.2.5 Déclarations de fonctions

Le langage **Yalta** accepte les déclarations de fonctions d'une manière proche du **langage C**. Seules les informations relatives aux types des variables disparaissent, et la valeur retournée, du fait de l'aspet *fonctionnel* de **Yalta**, devient obligatoire. La forme générale d'une déclaration de fonction est donc la suivante :

```
nom_fonction ( [ arg1 [ , arg2 [ ... ] ] ] )
{
    // Corps de la fonction,
    // C'est à dire suite d'expression ou de structures de contrôle...
    return ( expr );
}
```

Les éléments formant cette déclaration sont les suivants :

- `nom_fonction` : le nom de la fonction déclarée et sous lequel elle pourra être appelée,
- `[ arg1 [ , arg2 [ ... ] ] ]` : les noms des paramètres éventuels, ils ne sont pas indispensables (la fonction peut ne pas en avoir), ni même limités,

- `// Corps de la fonction` : le corps de la fonction peut être contitué d'expressions ou de structures de contrôle.
- `return ( expr )` : dans l'esprit fonctionnel de la programmation avec **Yalta**, une fonction doit nécessairement renvoyer une valeur, la valeur retournée l'est par cette *fonction* (en fait, un mot clé du langage), et la valeur renvoyée est celle de l'expression évaluée "`expr`".

Le nom d'un argument ne peut être le même que celui d'une constante.

5.5 Instructions de la machine virtuelle

La machine virtuelle accepte le code compilé pour l'exécuter. Elle dispose donc d'une panoplie complète et adaptée au problème, d'instructions manipulant les vecteurs de cotation. Elle agit sur plusieurs zones de mémoires distinctes :

- Une zone programme, où l'on trouvera le programme compilé,
- Une pile d'informations, pour faciliter le passage de paramètres aux fonctions câblées existantes, et celles qui seront définies par l'utilisateur,
- Des zones mémoires, pour stocker les différentes variables qui seront placées dans la pile pour être manipulées.

5.5.1 Liste des instructions

La forme générale de ses instructions est la suivante :

```
num_instruction arg1 arg2
```

C'est à dire un triplet comprenant :

1. `num_instruction` : le numéro identifiant l'instruction,
2. `arg1` : un premier argument,
3. `arg2` : un second argument.

5.5.1.1 Instructions de calcul

Ce jeu d'instructions est destiné aux calculs les plus élémentaires sur les vecteurs de cotation.

`OP_ALU` est le terme générique désignant ces opérations. D'un manière générale, ces opérations sont binaires. Elles prennent deux arguments sur la pile de données, et placent leur résultat au sommet de cette même pile. C'est le deuxième argument de l'instruction qui détermine l'opération effectuée.

La politique d'application de l'opération suivant les dates est la suivante :

- Pour deux vecteurs, l'opération demandée est appliquée termes à termes pour les dates correspondantes. Les dates n'apparaissant pas dans l'un ou l'autre des vecteurs sont ignorées pour le calcul, comme le montre la figure 5.2.
- Pour un vecteur et un singleton (i.e. : un vecteur à **un** élément), l'opération demandée est appliquée pour tous les termes du vecteur avec le singleton (c.f. : la figure 5.3), sans tenir compte de la date du singleton qui n'a de toute façon pas de valeur.
- Pour deux singletons, l'opération demandée est appliquée une fois et génère un autre singleton.

Les opérandes de cette instruction lui étant passés par la pile, elle prend pour premier opérande, l'avant dernier vecteur empilé, pour second opérande, le dernier vecteur empilé.

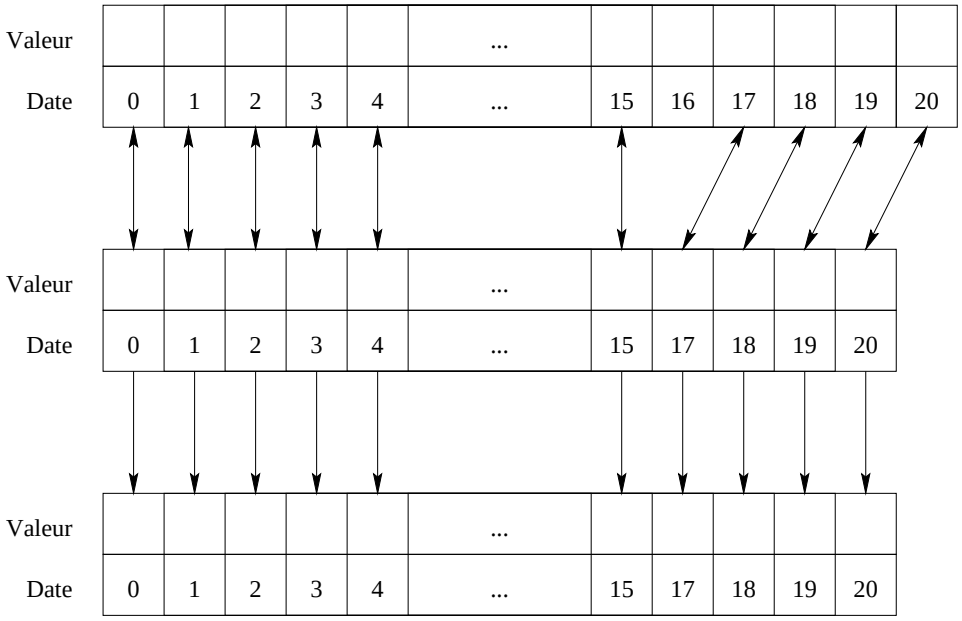

FIG. 5.2 – Yalta - Opération entre deux vecteurs.

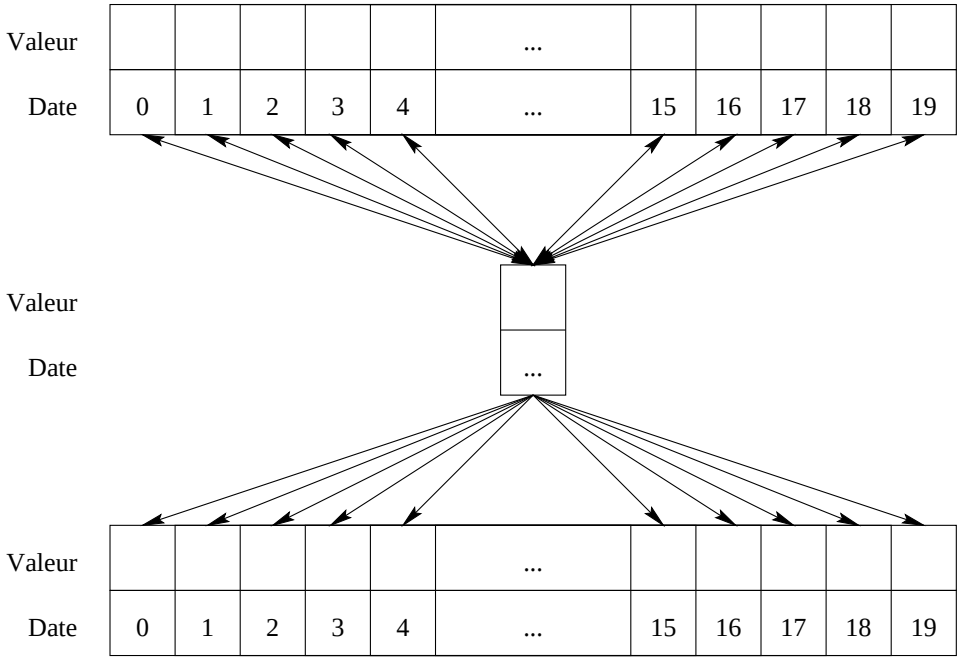

FIG. 5.3 – Yalta - Opération entre un vecteur et un singleton.

Les opérations disponible via le deuxième paramètre de `OP_ALU` sont les suivantes :

Argument de l'instruction	Description
Instructions booléennes	
`ALU_AND`	*ET* logique
`ALU_OR`	*OU* logique
`ALU_XOR`	*OU Exclusif* logique
`ALU_NOT`	*Non* logique **unaire**
`ALU_LT`	Inférieur strict
`ALU_GT`	Supérieur strict
`ALU_LE`	Inférieur ou égal
`ALU_GE`	Supérieur ou égal
`ALU_EQ`	Egal
`ALU_NE`	Différent
Instructions Arithmétiques	
`ALU_ADD`	Addition
`ALU_SUB`	Soustraction
`ALU_NEG`	Négatif **unaire**
`ALU_MUL`	Multiplication
`ALU_DIV`	Division
`ALU_MOD`	Modulo
`ALU_INC`	Incrémentation **unaire**

L'opération `ALU_INC` n'opère que sur la variable en haut de pile, et incrémente celle-ci, sans la sortir de la pile.

Les erreurs relatives à l'exécution de ces instructions, lèvent des exceptions spécifiques qui stoppent l'exécution du programme.

5.5.1.2 Instructions de manipulation de variables

Ces instructions sont des outils de manipulation des variables, à l'intérieur de la machine virtuelle. Elles permettent la création, l'initialisation et la copie de ces variables.

Les adresses auquelles les arguments de ces fonctions font références, sont celles générées par le compilateur, et valables pour l'environnement courant.

Brièvement :

Instruction	Arg1	Arg2	Description
`SCOPE`			Initialise un nouvel environnement
`VAR_INIT`	`VALUE`	`ADDRESS`	Initialise une variable
`VAR_STORE`	`ADDRESS`	`[NOPOP]`	Copie une variable depuis la pile

Dans le détail :

`SCOPE` :

Cette fonction permet la création d'un nouvel environnement. Elle initialise un nouvel environnement vide. Les variables y sont numérotées à partir de `0`. Un environnement est détruit à l'appel d'une instruction `RETURN` (c.f. : 5.5.1.3 page51).

`VAR_INIT` :

Cette instruction initialise la variable à l'adresse passée en second argument, en un singleton prenant la valeur (entière) du premier argument. La fonction n'a aucun

autre effet. La constante `NEW_VAR` passée en premier argument, est en fait `0`.

La fonction n'a aucun effet si la variable est déjà initialisée.

`VAR_STORE` :

Cette instruction copie le contenu du sommet de la pile, vers la variable à l'adresse passée en premier argument. Le sommet de la pile est dépilé par l'opération, sauf si un deuxième argument `NO_POP` est passé.

Remarque :

Le cas des constantes du langage, c'est à dire les vecteurs de cotation initiaux `o`, `h`, `l`, `c` et `v`, sont traités de la manière suivante :

- Leurs adresses sont négatives,
- Elle sont accessibles en lecture seulement (donc non modifiables).

5.5.1.3 Instructions de débranchement

Ces instructions permettent d'effectuer les débranchements relatifs aux structures de contrôle et aux appels de routines.

Les numéros d'instructions auxquels font référence les paramètres de ces instructions, correspondent au déplacement dans le fichier temporaire regroupant les instructions constituant le programme.

Instruction	Arg1	Arg2	Description
JUMP	NUM		Débranchement automatique
JUMPZ	NUM		Débranchement conditionnel
CALL	NUM		Appel de sous routine
RETURN			Retour de sous routine

`JUMP` :

Cette instruction provoque un débranchement automatique, non conditionnel, vers l'instruction dont le numéro est passé en paramètre.

`JUMPZ` :

`JUMPZ` a le même comportement que `JUMP`, excepté que le débranchement est conditionnel. Le débranchement est décidé sur le contenu du sommet de la pile (qui est, au passage, dépilé par l'instruction). Il y a débranchement si cette valeur est à `faux` (i.e. : c'est à dire `0`).

`CALL` :

L'instruction `CALL` est destinée à exécuter des appels de sous routines, donc les appels de fonctions de notre langage. Le numéro qu'elle accepte en argument, correspond au numéro de l'instruction débutant la routine dans le programme.

Cette instruction empile la position courante dans le programme (pour le retour de la routine), puis elle effectue le débranchement attendu.

`RETURN` :

C'est l'instruction de fin de routine. C'est cette instruction qui permet le retour d'une fonction dans notre langage, avec un passage de résultat.

La position précédemment empilée par l'instruction CALL, est récupérée pour le débranchement à effectuer pour le retour de la routine.

L'environnement courant, c'est à dire toutes les variables le concernant, est détruit.

Le débranchement est effectué.

5.5.1.4 Instructions de manipulation de pile

Ce jeu d'instructions est mis à la disposition du compilateur, pour générer les passages d'opérandes aux instructions arithmétiques et câblées (c.f. : OP_ALU, 5.5.1.1 page 48 et OP_BLT, 5.5.1.5 page 52), et aux fonctions utilisateurs. Ces différentes opérations effectuent des manipulations de pile, de la mémoire vers la pile, et de la pile vers la mémoire.

Instruction	Arg1	Arg2	Description
PUSH	METH	VAR	Empile une variable
POP_ARG	VAR		Dépile un argument
POP			Dépile un élément

PUSH :

L'instruction PUSH fonctionne en plaçant son deuxième argument en haut de la pile.

METH détermine la méthode utilisée pour l'*empilage*. METH est une constante, soit CONST, soit VAR :

- CONST indique que la valeur à empiler est une variable réelle, elle est alors construite et empilée à partir de l'instruction suivante qui n'est pas alors une instruction, mais le réel à empiler[6] ,
- VAR indique que le second argument est l'adresse de la valeur à empiler.

POP_ARG :

Cette instruction permet de placer en mémoire, les arguments qui sont en haut de la pile courante, après une instruction CALL. Le second argument passé à l'instruction est l'adresse (dans le nouvel environnement), à laquelle l'argument doit être stocké.

POP :

Cette instruction dépile un élément dans la pile, sans autre effet.

5.5.1.5 Instructions câblées

Ces instructions correspondent aux besoins spécifiques liés au problème traité : la manipulation de vecteurs de cotation.

OP_BLT est le terme générique désignant ces opérations. C'est le premier argument passé à l'instruction qui détermine l'opération effectivement appliquée.

Cette section ne détaille que quelques unes de ces instructions câblées, à titre d'exemple de fonctionnement.

Je donne trois exemples :

- BLT_LEN : Fonction permettant d'obtenir la longueur du vecteur,

[6] Une instruction PUSH CONST REEL tient donc l'espace de deux instructions, la seconde étant occupée par le réel.

- `BLT_REF` : Fonction permettant d'obtenir le vecteur référençant un vecteur n jours avant,
- `BLT_MM` : Fonction permettant d'obtenir la moyenne mobile d'un vecteur.

Au jour où ce rapport est rédigé, 10 autres instructions cablées spécifiques ont été implémentées :

- `BLT_CUM` : Fonction pour effectuer le cumul des valeurs d'un vecteur,
- `BLT_MOM` : Fonction calculant le momentum d'une cotation,
- `BLT_RSI` : Fonction calculant le RSI d'une cotation,
- `BLT_STD` : Fonction calculant une déviation standart,
- `BLT_HIG` : Fonction renvoyant la plus haute valeur du vecteur,
- `BLT_LOW` : Fonction renvoyant la plus basse valeur du vecteur,
- `BLT_HHV` : Fonction renvoyant la plus haute valeur du vecteur sur une période,
- `BLT_LLV` : Fonction renvoyant la plus basse valeur du vecteur sur une période,
- `BLT_SUM` : Fonction renvoyant la somme sur une période d'un vecteur,
- `BLT_OBV` : Fonction renvoyant l'indicateur "`obv()`", "On Balance Volume".

Fonctionnement général :

L'instruction est de la forme :

```
OP_BLT   BLT_FUN
```

Où `BLT_FUN` est l'instruction à exécuter (i.e. : `BLT_LEN`, `BLT_REF` ou `BLT_MM` dans les exemples qui suivent).

Les paramètres relatifs à la fonction qu'ils implémentent, sont construits par le compilateur de la même manière que pour les appels de fonctions de l'utilisateur, par `PUSH` (c.f. : 5.5.1.4 page 52), et sont dépilés par l'exécution de l'instruction.

Le résultat de l'instruction est placé en haut de la pile courante, après l'exécution.

Chacune de ces instructions implémente donc une fonction particulière du langage fournit à l'utilisateur, et spécifique au problème à traiter.

Les éventuelles erreurs relatives à l'exécution de ces instructions, lèvent des exceptions spécifiques qui stoppent l'exécution du programme.

`BLT_LEN :`

Cette instruction permet d'obtenir la longueur d'un vecteur. Elle implémente la fonction du langage : `len ( v )`. Elle renvoie en fait, le nombre de termes stockés dans le vecteur `v`.

Ainsi, une portion de langage utilisateur :

```
len ( v ) ;
```

Sera traduite :

```
0  PUSH    VAR      @V
1  OP_BLT  BLT_LEN
```

Le vecteur placé en haut de la pile à la fin de l'exécution de l'instruction, sera un singleton contenant la longueur du vecteur `v`.

`BLT_REF :`

C'est l'instruction qui déréférence un vecteur du nombre de jour demandé (c.f. : la figure 5.4 qui donne un exemple pour un déplacement de 2 jours). Elle implémente la fonction du langage : `ref ( v , -n )`, équivalente à `v [ n ]`.

Une portion de langage utilisateur :

```
v [ n ] ;
```

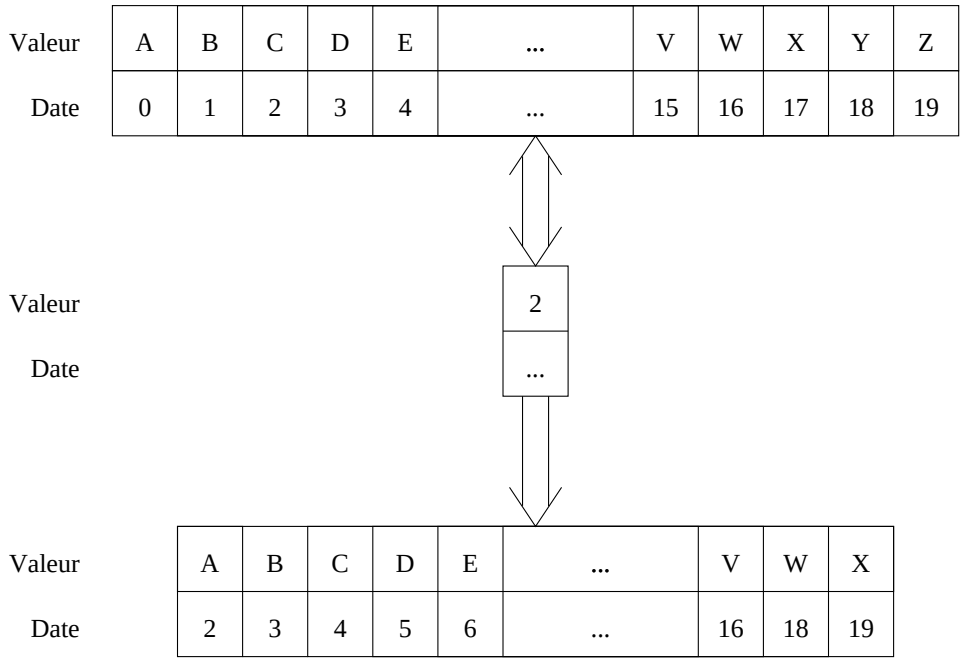

FIG. 5.4 – Yalta - Fonctionnement de l'instruction `BLT_REF`.

Prendra donc la traduction :

```
0  PUSH    VAR      @V
1  PUSH    VAR      @N
2  OP_BLT  BLT_REF
```

Le vecteur placé en haut de la pile à la fin de l'éxecution de l'instruction, sera le vecteur `v` déréférencé de `n` jour(s).

`BLT_MM :`

Cette instruction calcule une moyenne mobile sur un vecteur. Elle implémente la fonction du langage : `mov ( v , n , c )`[7].

Le code utilisateur :

```
mov ( v , 10 , S );
```

Sera compilé en :

```
0  PUSH    VAR      @V
1  PUSH    CONST    10
2  PUSH    CONST    11
3  OP_BLT  BLT_MM
```

Le vecteur placé en haut de pile à la fin de l'éxecution de l'instruction est alors la moyenne mobile *simple* à 10 jours du vecteur `v`.

5.5.2 Exemples de traductions

Les exemples suivants donnent des fonctions en langage utilisateur, et la traduction qu'elles prennent dans le langage de notre machine.

[7] c.f. : 5.3.2.2 page 40 pour l'explication sur les paramètres et 5.3.2.3 page 44 pour les constantes acceptées.

Les deux premiers exemples réimplémentent une fonction du langage qui existe *câblée* dans la machine virtuelle : la fonction `cum ( v )`[8]. Le premier exemple montre la possibilité d'appels récursifs, alors que le second donne une forme de la fonction avec une boucle.

Le troisième exemple donne une traduction pour deux boucles imbriquées.

5.5.2.1 Appel récursif

Le code utilisateur pour la fonction `cum ( a )` peut être :

```
cum ( a ) {
        self = if ( len ( a ) == 0 ,
                0 ,
                cum ( a [ 1 ] ) + a
        ) ;
        return ( self ) ;
}
```

[8] `cum ( v )` pour cumulate.

La traduction de ce code, pour la machine virtuel sera alors :

```
0   SCOPE
1   VAR_INIT    0         @A
2   POP_ARG     0
3   VAR_INIT    0         @SELF
4   PUSH        VAR       @A
5   OP_BLT      BLT_LEN
6   PUSH        CONST     0.000
8   OP_ALU      ALU_EQ
9   JUMPZ       13
10  PUSH        CONST     0.000
12  JUMP        20
13  PUSH        VAR       @A
14  PUSH        CONST     1.000
16  OP_BLT      BLT_REF
17  CALL        @CUM
18  PUSH        VAR       @A
19  OP_ALU      ALU_ADD
20  VAR_STORE   @SELF
21  PUSH        VAR       @SELF
22  RETURN
```

Après résolution des appels de fonction, ou *link*, l'appel récursif de la ligne `15` sera résolu en :

```
15  CALL       1
```

La ligne `1`, peut-être assimilée à une déclaration de variable. Elle est locale et sera détruite par le `RETURN` ligne `20`.

5.5.2.2 Boucle

Le code utilisateur de la fonction `cum ( v )`, aurai pu être implémenté aussi à l'aide d'une boucle, de la manière suivante :

```
cum2 ( a ) {
        self ;
        for ( i = 0 ; i <= len ( a ) ; i++ ) {
                self = self + a [ i ] ;
        }
        return ( self ) ;
}
```

La traduction d'un tel code serait alors :

```
0   SCOPE
1   VAR_INIT    0         @A
2   POP_ARG     @A
3   VAR_INIT    0         @SELF
4   PUSH        VAR       @SELF
5   POP
6   VAR_INIT    0         @I
7   PUSH        CONST     0.000
9   VAR_STORE   @I
10  PUSH        VAR       @I
11  PUSH        VAR       @A
12  OP_BLT      BLT_LEN
13  OP_ALU      ALU_LE
14  JUMPZ       27
15  JUMP        20
16  PUSH        VAR       @I
17  OP_ALU      ALU_INC
18  VAR_STORE   2         NOPOP
19  POP
20  JUMP        10
21  PUSH        VAR       @SELF
22  PUSH        VAR       @A
23  PUSH        VAR       @I
24  OP_BLT      BLT_REF
25  OP_ALU      ALU_ADD
26  VAR_STORE   @SELF
27  JUMP        16
28  PUSH        VAR       @SELF
29  RETURN
```

La possibilité d'optimiser le calcul de la valeur butoir pour la boucle se fait *naturellement* sentir, puisque avec une telle compilation, le code "`len ( v )`" est évalué à chaque itération (il se traduit ligne `6` et `7`). Cependant cette méthode de compilation pour une boucle permet de rendre la borne supérieure variable au cours des itérations, ce qui peut aussi présenter tout son intérêt.

De même, on se rend compte que les instructions `18` et `19` pourraient être ramenées, en supprimant le second argument de l'instruction `18`, à une simple instuction :

```
18  VAR_STORE   2
```

Mais le compilateur génère ce code pour le cas général où l'expression traduite (en l'occurence "`i++`"), se trouverait elle même opérande d'une autre expression.

5.5.2.3 Deux boucles imbriquées

Le code dont proposé ici n'a pas de sens particulier. Il s'agit simplement d'un exemple comportant une boucle imbriquée dans une autre.

Le code utilisateur est :

```
essais () {
        a = 0 ;
        for ( i = 0 ; i < 5 ; i++ ) {
                for ( j = 0 ; j <= 10 ; j++ ) {
                        a = a + i * j ;
```

```
                }
            }
            return ( a ) ;
    }
```

Et la traduction de ce code sera :

```
0   SCOPE
1   VAR_INIT    0         @A
2   PUSH        CONST     0.000
4   VAR_STORE   @A
5   VAR_INIT    0         @I
6   PUSH        CONST     0.000
8   VAR_STORE   @I
9   PUSH        VAR       @I
10  PUSH        CONST     5.000
12  OP_ALU      ALU_LT
13  JUMPZ       43
14  JUMP        20
15  PUSH        VAR       @I
16  OP_ALU      ALU_INC
17  VAR_STORE   @I        NOPOP
18  POP
19  JUMP        9
20  VAR_INIT    0         @J
21  PUSH        CONST     0.000
23  VAR_STORE   @J
24  PUSH        VAR       @J
25  PUSH        CONST     10.000
27  OP_ALU      ALU_LE
28  JUMPZ       42
29  JUMP        35
30  PUSH        VAR       @J
31  OP_ALU      ALU_INC
32  VAR_STORE   @J        NOPOP
33  POP
34  JUMP        24
35  PUSH        VAR       @A
36  PUSH        VAR       @I
37  PUSH        VAR       @J
38  OP_ALU      ALU_MUL
39  OP_ALU      ALU_ADD
40  VAR_STORE   @A
41  JUMP        30
42  JUMP        15
43  PUSH        VAR       @A
44  RETURN
```

5.6 Compilateur

Le compilateur permet l'obtention de code binaire dans lequel les appels de fonctions (`CALL`) ne sont pas résolus. Le compilateur fonctionne en une passe comprenant l'analyse lexicale, l'analyse syntaxique et la gestion des symboles de variables et de fonctions.

Les fichiers sources du compilateur se trouvent dans le répertoire "`comp`".

5.6.1 Principe général et utilisation

Le nom de l'exécutable correspondant au compilateur est : "`yalta_compiler`". Son utilisation est la suivante :

```
yalta_compiler  -o file_out  [-i file_in]  [-e file_err]
```

Où :

- `file_out` : est un argument nécessaire pour nommer le projet, un fichier "`file_out.yc`"[9] et un fichier "`file_out.yl`"[10] seront produits par le compilateur,
- `file_in` : est le nom du fichier à compiler, si cet argument est omis, le code à compiler sera attendu sur l'entrée standart,
- `file_err` : est le nom du fichier d'erreurs dans lequel les messages issus de la compilation seront écrits, si cet argument est omis, les messages seront écris dans le fichier d'erreurs standart.

Le compilateur renvoie les codes d'erreurs suivant (cf : "`const.h`") :

- `1` : arguments invalides,
- `2` : erreurs internes, pour le développement uniquement,
- `3` : erreurs fatales à l'émission de code compilé,
- `4` : erreurs de la pile des symboles ou des fonctions, si elles sont pleines,
- `5` : erreurs de syntaxe, le parser a détecté une erreur.

Toutes les erreurs sont accompagnées d'un message dans le fichier d'erreurs avec le numéro de ligne atteint et le dernier token trouvé si possible.

5.6.2 Analyseur lexical

L'analyse lexicale (voir le fichier "`lex.c`") aurait pu être implémentée par un automate généré par l'un des utilitaires existants pour créer un tel outil (*lex*, *flex*...). Mais la solution apportée au problème pour cette analyse lexicale, a été construite *de toutes pièces*.

Pour rester compatible avec *Yacc*, la fonction remplissant le rôle d'analyseur lexical s'appelle "`yyparse`". C'est un simple automate déterministe, analysant avec un caractère d'avance le fichier d'entrée, et renvoyant un entier correspondant au token reconnu.

D'une manière générale :

- Les chaînes de caractères sont des symboles de type indéterminé, parmi lesquels sont reconnus les mots clés et les constantes du langage, par une recherche dichotomique dans une table particulière "`keytabl`",
- Les entiers et les réels sont reconnus par l'analyseur, et retournés à l'analyseur syntaxique, avec leur valeur correspondante, comme des réels,
- Les opérateurs standards sont reconnus, et renvoyés comme des *tokens* distincts.

[9] `.yc` : Yalta Code.
[10] `.yl` : Yalta Link.

5.6.3 Analyseur syntaxique

L'analyseur syntaxique (voir le fichier "`parser.y`") est construit grâce à l'utilitaire *Yacc*. La grammaire utilisée est donc **LALR(1)**[11].

Elle reconnait le langage **Yalta** et le transforme donc dans le pseudo assembleur qui lui est dédié. Cette phase correspondant à la compilation, se fait parallèlement à l'analyse lexicale, dans une même et unique passe.

L'analyseur inclu la gestion d'une pile indépendante de l'automate d'analyse **LALR(1)**, pour résoudre les débranchements relatifs aux instructions de contrôle (`FOR`, `WHILE`...), par retour arrière.

Une autre pile est utilisée pour stocker et générer la liste des arguments passés aux fonctions utilisateurs.

5.6.4 Gestion des symboles de variables

Le compilateur gère les variables, fonctions par fonctions, *scope* par *scope* (voir le fichier "`symbol.c`"). Chaque nouvelle variable déclarée entraîne l'émission d'une instruction du type :

```
VAR_INIT    0     @ADRESSE
```

Où l'adresse choisie est la première disponible dans l'environnement courant.

L'utilisation d'une variable portant le même nom qu'une fonction entraîne une alerte à la compilation. Mais cette alerte n'est levée que si la fonction est définie en amont, dans le même fichier.

5.6.5 Gestion des symboles de fonctions

Le compilateur ne résout aucun débranchement relatif aux appels de fonctions (ou `CALL`), cette tâche étant remplie par le lieur. Par conséquent, la gestion des symboles de fonction est destinée à produire les informations qui seront nécessaires au lieur (voir le fichier "`linktalb.c`"), c'est à dire à la récolte des informations qui seront incrites dans le fichier "`file_out.yl`".

Deux tables de fonctions sont maintenues à jour, les fonctions définies dans le fichier en cours d'analyse, et les fonctions appelées depuis le fichier en cours.

Pour chacune des ces fonctions, les informations suivantes sont stockées, puis émises dans le fichier "`file_out.yl`" :

- le nom de la fonction,
- le nombre de paramètres,
- le numéro d'instruction où commence la routine dans le code compilé,
- le numéro de ligne où a lieu l'appel ou la définition dans le fichier d'origine.

Ces routines permettent de détecter, dans un même fichier, les définitions redondantes de fonctions et les appels de fonctions définies avec un nombre invalide d'argument.

La taille du code émis, en nombre d'instruction, est elle aussi insérée dans le fichier "`file_out.yl`".

5.7 Lieur

Le lieur ou "*linker*" est l'utilitaire qui permet d'obtenir du code exécutable à partir des portions de code issues du compilateur.

Le lieur fonctionne en récoltant dans un premier temps, les informations relatives aux fonctions définies et appelées dans les différents projets, grâce aux fichiers "`.yl`".

[11] **LALR(1)** : LookAhead Left Recursive avec 1 token d'avance.

Dans un second temps, il réémet le code correspondant à la *somme* des différents projets, en y résolvant les appels de fonctions.

Les fichiers sources du lieur se trouvent dans le répertoire "`linker`".

5.7.1 Principe général et utilisation

Le nom de l'exécutable correspondant au lieur est : "`yalta_linker`". Son utilisation est la suivante :

```
yalta_linker -o file_out [-e file_err] project1 [project2 [...]]
```

Où :

- `file_out` : est un argument nécessaire pour nommer le fichier *exécutable* qui sera généré sous le nom "`file_out.ylc`"[12] ,
- `file_err` : est le nom du fichier d'erreurs dans lequel les messages issus de la compilation seront écris, si cet argument est omis, les messages seront écris dans le fichier d'erreurs standart,
- `projet`$_i$: sont les noms des projets à lier entre eux. Au moins un nom de projet est requis pour l'exécution du lieur. Pour chacun des "`projet`$_i$", un fichier "`projet`$_i$`.yc`" et un fichier "`projet`$_i$`.yl`" sont attendus.

Le lieur renvoie les codes d'erreurs suivants (c.f. : "`const.h`") :

- `1` : arguments invalides,
- `2` : fichier introuvable ou impossible à ouvrir,
- `3` : fichier "`.yl`" invalide,
- `4` : définition redondante d'une fonction,
- `5` : appel de fonction non résolu,
- `6` : appel de fonction avec un nombre invalide de paramètres,
- `7` : erreur lors de l'émission du code compilé et lié,
- `8` : point d'entrée (fonction "`__main__`") introuvable.

Toutes les erreurs sont détaillées dans le fichier d'erreurs avec le numéro de ligne et le nom du (des) projet(s) ayant déclenché l'erreur, ainsi que le nom de la fonction pour laquelle ladite erreur a eu lieu.

5.7.2 Résolution des appels de fonction

La résolution se fait, par un inventaire des différentes déclarations et des différents appels de fonctions (voir le fichier "`link_tabl.c`"). *Projets* par *projets*, les fichiers "`projet`$_i$`.yl`" sont parcourus, et deux tables de fonctions sont construites.

La table "`def_func`" fait un inventaire des fonctions définies, alors que la table "`nes_func`" fait un inventaire des fonctions appelées.

Les labels, ou points d'entrées des fonctions dans le code, sont résolus avec leurs adresses absolues dans le code final.

Ces routines permettent de détecter les erreurs suivantes :

- Les fichiers "`projet`$_i$`.yl`" invalides ou comportant des erreurs,
- Les définitions des fonctions redondantes,
- Les appels de fonctions non résolus,
- Les appels de fonctions avec un nombre invalide de paramètres.

5.7.3 Résolution du point d'entrée du programme

Le label correspondant au point d'entrée du programme, la fonction "`__main__`", subi un traitement particulier. Ce label est détecté lors de la résolution des appels de fonctions.

[12] `.ylc` : Yalta Linked Code.

Si ce point n'est pas trouvé, le lieur n'émet aucun code, sinon, le code émis commence par un saut inconditionnel "`JUMP`" vers le point d'entrée de cette fonction.

5.7.4 Réémission du code exécutable

La réémission du code exécutable se fait en deux temps (voir le fichier "`emit.c`").

Dans un premier temps, les codes non liés des différents projets "$\texttt{projet}_i\texttt{.yc}$", sont simplement concaténés dans le fichier "`file_out.ylc`", sans plus de vérification.

Dans un second temps, la table "`nes_func`" des fonctions appelées depuis les différents projets, est parcourue, et chaque appel de fonction donne lieu à un retour arrière dans le code. Ainsi, les appels de routines "`CALL`" sont résolus.

5.8 La machine virtuelle

La machine virtuelle est l'outil final, exécutant le programme compilé et lié. Elle charge tout d'abord un fichier de données, correspondant l'historique des valeurs de la cotation à analyser. Elle charge ensuite le programme compiler, puis exécute "*à la volée*" les instructions fournies par ce programme.

Les fichiers sources du lieur se trouvent dans le répertoire "`yvm`".

5.8.1 Principe général et utilisation

Le nom de l'exécutable correspondant à la machine virtuelle est : "`yalta_vm`". Son utilisation est la suivante :

```
yalta_vm        -i file_in -v values_in
```

Où :

- `file_in` : est un argument nécessaire pour nommer le fichier *exécutable* à charger, le nom "`file_out.ylc`" sera cherché,
- `values_in` : est un argument nécessaire pour nommer le fichier de *valeurs* à charger,

La machine virtuelle renvoie les codes d'erreurs suivants (c.f. : "`const.h`") :

- `1` : arguments invalides,
- `2` : fichier introuvable ou impossible à ouvrir,
- `3` : erreur mémoire,
- `5` : erreur à l'éxecution (instruction invalide),

Toutes les erreurs sont détaillées sur la sortie standard d'erreurs.

5.8.2 Formatage des entrées sorties

Le fichier d'entrée doit se tenir à une syntaxe très simple, mais particulière. Les sorties de la machine virtuelle, c'est à dire le résultat de l'exécution du programme compilé, se tienne à cette même syntaxe.

5.8.2.1 Fichier de valeurs

Les routines chargeant le fichier de valeurs sont disponibles dans le fichier "`dvread.c`". Le fichier de valeur doit permettre de créer les vecteurs de cotations `O`, `H`, `L`, `C` et `V` de la machine virtuelle, tous liés par les dates de ces valeurs. La syntaxe adoptée est la suivante :

```
YYYY-MM-DD   O.OO   H.HH   L.LL   C.CC   V
```

Chaque champ étant séparé par une tabulation.

5.8.2.2 Sortie

Les vecteurs affichés dans le fichier de sortie prennent le même type de syntaxe, sauf qu'un seul réel est renseigné pour une date. Les différents vecteurs sont séparés par une ligne vide :

```
YYYY-MM-DD   X.XX
YYYY-MM-DD   X.XX
YYYY-MM-DD   X.XX

YYYY-MM-DD   Y.YY
YYYY-MM-DD   Y.YY
YYYY-MM-DD   Y.YY
```

Cet exemple donne le type de sortie généré pour deux vecteurs de taille 3.

5.8.3 Type de données

Le type abstrait de données manipulé par la machine virtuelle (définit avec les autres types dans le fichier "`const.h`") est le suivant :

```
/* day/vector data structure */
typedef struct {
        int size;
        int *date;
        double *value;
} dvector_t;

#define DVECTOR_Z sizeof ( dvector_t )
```

Le fichier "`dvector.c`" donne toutes les routines élémentaires pour la manipulation de ce type.

5.8.4 Chargement du programme

Le fichier "`loader.c`" contient les routines effectuant le chargement en mémoire du programme. Il est en fait simplement chargé dans un tableau d'instructions dans lequel il sera aisé ensuite d'effectuer les débranchements voulus.

5.8.5 Exécution du programme

Toute la partie *exécution* du programme est gérée par les routines fournies par le fichier "`run.c`". C'est là que se trouve le traitement effectué à la rencontre des différentes instructions.

C'est, *grossièrement*, un automate déterministe gérant une pile de donnée (les adresses des vecteurs) et de débranchements.

5.8.6 Bibliothèques de fonctions cablées

Deux principales bibliothèques de fonctions, pour le type "`dvector`", implémentent toutes les instructions mis à disposition par la machine virtuelle. Il s'agit de :

- "`alu.c`" implémente toutes les instructions élémentaires "`OP_ALU`",
- "`blt.c`" implémente toutes les instructions spécifiques à l'analyse technique "`OP_BLT`".

Ces fonctions sont souvent des adaptations à la structure "`dvector`" de portions de codes existants.

5.9 Bibliothèque de fonctions programmée

Il n'est pas paru indispensable d'implémenter toutes les fonctions à programmer "*en dur*" **dans** la machine virtuelle. Une partie des ces fonctions, celles qui découlaient simplement de celles qui étaient déjà *cablées*, a été programmée dans le langage **Yalta** lui-même.

Ainsi, le fichier "`ylib`" contient le code source de la bibliothèque "`ylib.yc`" qu'il convient de lier avec son propre programme pour pouvoir disposer de la totalité des fonctions implémentées par **Yalta**.

Ceci donne, par exemple :

```
ad()
{
        return (cum((((c - l) - (h - c)) / (h - l)) * v));
}
stochmomentum(t1, t2, t3)
{
        tllv = llv(l, t1);
        thhv = hhv(h, t1);

        return (100 * (mov(mov(c - (0.5 * (thhv + tllv)), t2, e), t3, e) /
                       mov(mov(thhv - tllv, t2, e), t3, e)));
}
```

Au total, une vingtaine de fonctions sont programmées ainsi.

5.10 Autres outils

Quelques autres outils ont étés développés autour de **Yalta**. En particulier :
- un petit "*désassembleur*" nommé "`uas`",
- mais aussi, et surtout **javaGrapher**, une application **Java** permettant l'affichage des résultats de **Yalta**, sous forme graphique, et développée par **Arnaud Lauret**, lui-même stagiaire dans l'entreprise pendant cette période.

Pour donner une vue de cette application graphique, voiçi un programme :

```
__main__()
{
  macd_std_indicator();
  llv(macd_std_indicator() ,50);
  hhv (c, 50);
  llv(c , 50);
  c;
  return (0);
}
```

Ce programme calcul pour une cotation :
- l'indicateur MACD[13] standard,
- une indication du support de cet indicateur sur une période de 50 jours (i.e. : la plus base valeur sur les 50 derniers jours),
- une indication de la résistance à la clôture (close) sur cette même période (i.e. : la plus haute valeur sur les 50 derniers jours),
- une indication du support à la clôture (close) sur cette même période.

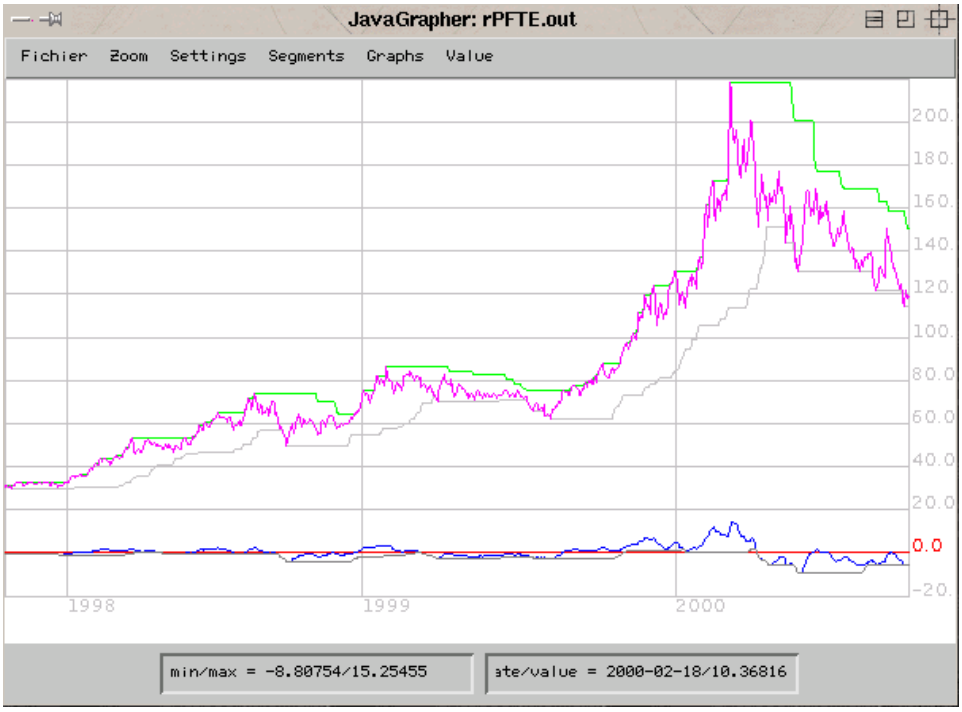

FIG. 5.5 – Yalta - Vue de **javaGrapher**

Le résultat de **Yalta**, exécuté avec ce programme, sur les valeurs de ces 3 dernières années de la cotations **France Télécom**, et présenté par **javaGrapher**, donne le résultat présenté en figure 5.5 (capture d'écran sous **Linux**).

[13]MACD : Moving Average Convergence/Divergence indicator.

Chapitre 6

Conclusion

Le stage que j'ai effectué, et les conditions dans lesquelles je l'ai effectué ont été particulièrement bénéfiques à l'évolution de ma *culture informatique*.

S'il est certain que j'avais déjà quelques notions appronfondies sur quelques-uns des problèmes qu'il m'a été donné de traiter, il est aussi certain que ce n'est que pendant ce stage, que j'ai pu mettre en oeuvre ces compétences, les enrichir par l'expérience de toutes les personnes qui m'ont aidé pendant ce stage, et les expérimenter.

Je tiens donc particulièrement à remercier, les personnes suivantes :

- Jean-Pierre Bretaudière, mon maître de stage,
- Pascal Nicolas, chargé du suivi de ce stage pour l'Université.

Mais aussi : Thierry Meyer, Arnaud Lauret, Anselme Préhaut, Christian Gandon, et tout le reste de l'équipe à Unimédia et Angers Journal.

Bibliographie

[1] Linux Man Pages, écrites pas les différents dévellopeurs du système d'exploitation Linux.

[2] Pthread Programming, Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrel, O'Reilly, 1996.

[3] UNIX Network Programming Volume 1, Networking APIs Sockets and XTI, W. Richard Stevens, Prentice Hall, 1998.

[4] UNIX Network Programming Volume 2, Interporcess Communication, W. Richard Stevens, Prentice Hall, 1999.

[5] Programmer avec les Threads UNIX, Charles J. Northrup, John Wiley & Sons, 1996.

[6] MySQL Reference Manual for version 3.23.18-alpha, TcX AB & MySQL Finland AB & Detron HB Stockholm SWEDEN, Helsingfors FINLAND and Uppsala SWEDEN, 1996.
Ce manuel peut être trouvé, sous sa forme "`html`", dans le fichier "`/usr/local/mysql/Docs/manual.html`".

[7] The Unix System, S.R. Bourne, Addison-Wesley, 1982.

[8] Reuters Financial News & Selectfeed Plus User Guide, Reuters, 1996.

[9] MySQL & mSQL, Randy Jay Yarger, George Reese & Tim King, O'Reilly, 1999.

[10] Introduction to Compiler Construction with Unix, Axel T. Schreiner, H. George Friedman, Jr, Prentice-Hall, 1985.

[11] Compiler Design and Construction, 2nd edition, Arthur B. Pyster, Van Nostrand Reinhold, 1980.

[12] Lex & Yacc, John R. Levine, Tony Mason & Doug Brown, O'Reilly, 1994.

[13] MetaStock User's Manual, Equis, 1997.

[14] Beginning Linux Programming, Neil Matthew & Richard Stones, Wrox Press, 1996.