GNU LINUX MAGAZINE / FRANCE

L 19275 - 91 - F: 6,20 €

France Métro : 6,20€ - DOM 6,75€ - TOM 950 XPF - BEL : 6,80€ - LUX : 6,80€ - PORT. CONT. : 6,80€ - CH : 12,70CHF - CAN : 11,60$ - MAR : 70DH

▶FÉVRIER ▶2007 ▶NUMÉRO 91

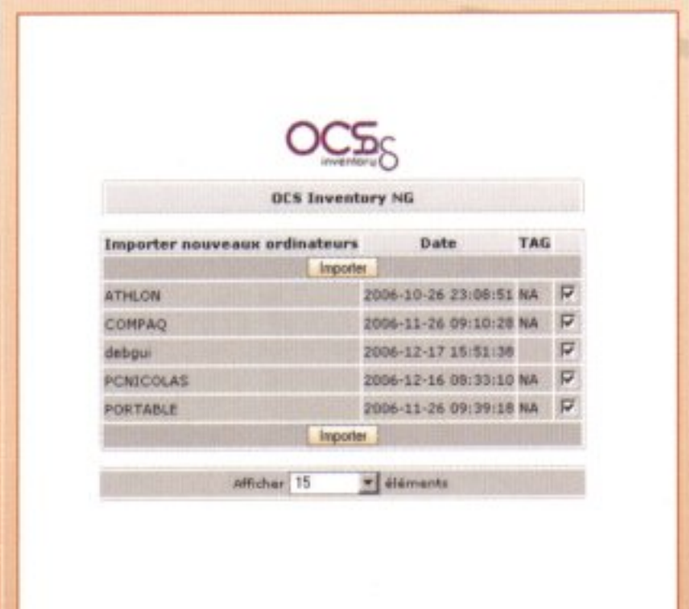
AJAX AVANCÉ

Principe, fonctionnement et pièges

p. 30

AJAX (Asynchronous Javascript And XML) est un terme à la mode décrivant simplement l'utilisation conjointe de quelques technologies éprouvées. Pourtant, derrière cet acronyme, se cachent de vrais besoins et des solutions concrètes. Découvrez ce qu'est vraiment AJAX et quelles en sont les limites.

SUPERVISION et GESTION



▶ Déployez OCS Inventory NG et GPLI

Assurez-vous la maîtrise de votre parc informatique en confiant l'inventaire à OCS Inventory NG et la gestion à GPLI. Le tutoriel proposé vous permettra de rapidement faire fonctionner ces deux applications de concert.

p. 52

Administration et développement sur systèmes UNIX

Edito

Bonjour et bienvenue dans ce nouveau numéro !

AJAX ! Non, pas le produit ménager ! Je vous parle du *buzzword* AJAX, l'acronyme d'*Asynchronous JavaScript and XML*. Qui ne parle pas en ce moment d'AJAX et du Web 2.0. Si vous avez un peu cherché par vous-même, vous vous êtes sans doute rendu compte qu'il n'y avait rien de nouveau dans cette technologie, qui n'en est d'ailleurs pas vraiment une. Cela se résume simplement à l'échange de données XML de manière asynchrone entre un client Web et le serveur. Bien entendu, j'ai bien dégrossi l'ensemble, mais la base se résume à cela.

Ce qui est important dans AJAX, ce n'est pas tant les technologies utilisées, mais la migration en douceur de bon nombre de projets. On assiste ainsi à une sorte de révolution en douceur, une évolution. Une fois le dialogue asynchrone entre client et serveur Web largement popularisé, la voie se tracera d'elle-même. Les applications dans leur grande majorité migreront en ligne. Certains éditeurs nous vendent déjà ainsi leurs solutions : pas de maintenance, pas de mise à jour, pas de problème de disponibilité...

Bien entendu, il nous manque encore des briques pour en arriver là, mais c'est clairement ce qui arrivera dans les années qui suivent. La suite est tout aussi évidente. Si les applications ont migré, pourquoi ne serait-ce pas le cas des données. Le discours marketing est le même, sécurité des données, disponibilité, personnalisation : « Votre maison brûle, vous ne perdez rien. Allez chez des amis, tout est là, photos, vidéos, rendez-vous, contrat, gestion, etc. ».

Nous assisterons à la dématérialisation de nos données privées et à la création d'espaces personnels sur la toile. Mais cette abstraction avec le matériel ou la situation géographique comporte des risques. Je parle de sécurité des données et de protection de la vie privée. Les solutions seront-elles à la hauteur dans le domaine ? La réponse est simple : oui, pendant un temps, comme toujours. Je pense donc que nous devrions assister à la fin de l'informatique personnelle au bénéfice du « tout terminal ». Mais le temps nous le dira...

AJAX est une première étape qui, finalement, va au-delà du buzzword. Les problématiques sont réelles et les solutions aussi, comme vous le constaterez par vous-même dans l'article consacré à ce sujet.

Je vous donne rendez-vous le 24 février prochain pour le numéro 92.

Denis Bodor

GNU Linux Magazine
est édité par Diamond Editions
B.P. 20142 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail :
lecteurs@gnulinuxmag.com
Service commercial :
abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication :
Arnaud Metzler
Rédacteur en chef :
Denis Bodor
Mise en page :
Fabrice Krachenfels
Responsable publicité :
Véronique Wilhelm
Tél. : 03 88 58 02 08
Service abonnement :
Tél. : 03 88 58 02 08
Relecture :
Dominique Grosse

Impression :
VPM Druck Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Distri-médias :
Tél. : 05 61 72 76 24

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution /N° ISSN : 1291-78 34
Commission Paritaire : 09 08 K78 976
Périodicité : Mensuel
Prix de vente : 6,20 €

WWW.GNULINUXMAG.COM

Denis Bodor

Aptitude, une interface à la mode pour APT

La toute récente diffusion du numéro hors série consacré aux distributions Debian a soulevé quelques critiques, parmi lesquelles, le fait qu'il n'ait pas été fait mention d'Aptitude dans l'article traitant des outils de gestion de paquets. Corrigeons cela avec une petite présentation ici-même.

Aptitude peut être vu comme le remplaçant de l'horrible `dselect`. Il se veut aisé d'utilisation tout en restant intéressant pour un utilisateur avancé. Je ne vous cache pas que je ne suis pas pleinement convaincu. Coté administrateur système, les outils `apt*` en ligne de commande ont l'énorme avantage d'être disponibles par défaut. L'utilisateur lambda, quant à lui, préfèrera sans doute un outil avec une GUI comme Synaptic.

Quoi qu'il en soit, l'exécution sans option de la commande `aptitude` nous donne ceci :

```
 Actions  Annuler  Paquet  Solutions  Rechercher  Options  Vues  Aide
C-T : Menu  ? : Aide  q : Quitter  u : M-à-J  g : Téléch./Inst./Suppr. Pqts
aptitude 0.4.4
--- Paquets pouvant être mis à jour
--\ Paquets installés
  --\ admin - Utilitaires d'administration (installation de logiciels, gestion d
    --\ main - L'archive principale de Debian
i     alien                                    8.64       8.64
i     apt-show-source                          0.10       0.10
i     apt-show-versions                        0.25.1     0.25.1
i     apt-src                                  0.4.4-1    0.4.4-1
      aptitude                                 3.1.10     3.1.10
i     at
terminal-based apt frontend                                          #
aptitude is a terminal-based apt frontend with a number of useful features,
including: a mutt-like syntax for matching packages in a flexible manner,
dselect-like persistence of user actions, the ability to retrieve and display
the Debian changelog of most packages, and a command-line mode similar to that
of apt-get.

aptitude is also Y2K-compliant, non-fattening, naturally cleansing, and
housebroken.

Marqueurs: admin::configuring, admin::package-management, implemented-in::c++,
```

Comme vous pouvez le constater, l'interface en mode texte (LibnCurses) offre une certaine ergonomie. Mais ce n'est pas là le point fort de cet outil. En effet, c'est en ligne de commande qu'Aptitude offre certaines facilités qui peuvent être appréciées. Les opérations habituelles d'installation et de désinstallation de paquets sont similaires aux manipulations faites avec `apt-get` (`aptitude install paquet` ou `aptitude remove paquet`). On notera l'existence d'une action `purge` équivalente à l'option `--purge` de l'action `remove` d'`apt-get`.

Les dépendances sont gérées par le système APT, mais Aptitude offre une interface un peu plus explicite quant à la résolution des conflits et des dépendances. Ainsi :

```
# aptitude purge bc
Construction de l'arbre des dépendances... Fait
Lecture de l'information d'état étendu
Initialisation de l'état des  paquets... Fait
Construction de la base de données des étiquettes... Fait
Les paquets suivants sont CASSÉS:
  mjpegtools
[...]
Les paquets suivants seront ENLEVÉS:
  bc{p}
0 paquets mis à jour, 0 nouvellement installés,
1 à enlever et 733 non mis à jour.
Il est nécessaire de télécharger 0o d'archives.
Après dépaquetage, 193ko seront libérés.
Les paquets suivants ont des dépendances non satisfaites:
```

cgiirc
IRC via le Web

Le Web est utilisable partout, depuis toutes les connexions, au travers de tous les *firewalls* (ou presque) et avec n'importe quel système, ce qui n'est pas le cas de l'IRC. Pour faciliter les choses, `cgiirc` se propose tout simplement de faire la passerelle entre les deux mondes.

Comme son nom l'indique, `cgiirc` est un script CGI permettant l'accès à un serveur IRC préconfiguré (ou non) et un canal préconfiguré (ou non). Attention, il faut bien comprendre que c'est un client IRC et non un serveur de connexion. Bien qu'il soit possible de le configurer de manière à ouvrir plusieurs connexions, l'ergonomie n'est pas celle d'une *applet* Java de « tchat » comme on en trouve en bonus sur certains sites Web.

`cgiirc` conviendra parfaitement, par exemple, en guise de solution temporaire universellement accessible. Une autre application est l'utilisation en tant que partie cliente pour une infrastructure intranet. Il suffira d'installer un serveur IRC sur une machine dédiée et le tour est joué. Un minimum de formation des utilisateurs sera nécessaire, mais les avantages face à une solution de type Jabber sont évidents : indépendance du client et interface Web.

En termes de sécurité, veillez à correctement configurer le serveur HTTP et, au besoin, à filtrer un maximum les connexions et les échanges. Un CGI reste un CGI.

Le BTS contient une seule entrée ouverte de niveau Wishlist et l'information est aux mains des développeurs amonts : « make it easier to add new smilies ».

```
mjpegtools: Dépend: bc mais il n'est pas installable
Resolving dependencies...
Les actions suivantes permettront de résoudre ces dépendances:

Supprimer les paquets suivants:
mjpegtools
Laisser les dépendances suivantes non satisfaites:
transcode recommande mjpegtools
Le score est de -501

Accepter cette solution? [Y/n/q/?]
```

Comme vous pouvez le voir, `mjpegtools` dépend de `bc` (on se demande pourquoi d'ailleurs). En supprimant `bc`, on « casse » `mjpegtools` qui doit donc être désinstallé. `apt-get` vous aurait renseigné de la sorte :

```
# apt-get remove --purge bc
[...]
Les paquets suivants seront ENLEVÉS:
  bc* mjpegtools*
0 mis à jour, 0 nouvellement installés,
2 à enlever et 732 non mis à jour.
Il est nécessaire de prendre 0o dans les archives.
Après dépaquetage, 1573ko d'espace disque seront libérés.
Souhaitez-vous continuer [O/n]?
```

Aptitude est donc plus explicité, mais, surtout, il apporte un petit quelque chose qui peut être intéressant (mis en évidence en rouge). Il nous informe, en effet, que `mjpegtools` est un paquet recommandé par `transcode`. Il s'agit d'une dépendance `Recommends`. En confirmant la solution proposée par Aptitude, les dépendances sont résolues comme souhaité :

```
[...]
Les paquets suivants seront ENLEVÉS:
  bc{p} mjpegtools
0 paquets mis à jour, 0 nouvellement installés,
2 à enlever et 732 non mis à jour.
Il est nécessaire de télécharger 0o d'archives.
Après dépaquetage, 1573ko seront libérés.
Voulez-vous continuer? [Y/n/?]
```

A l'installation de paquet, Aptitude montre également un léger changement vis-à-vis des dépendances :

```
[...]
Les NOUVEAUX paquets suivants vont être installés:
  libfaad0
Les paquets suivants seront mis à jour:
  libfaad2-0 liblame0 transcode
Les paquets suivants sont RECOMMANDÉS mais ne
seront pas installés:
  mjpegtools toolame transcode-doc
3 paquets mis à jour, 1 nouvellement installés,
0 à enlever et 729 non mis à jour.
[...]
```

Un autre élément caractérise Aptitude : la gestion des paquets installés automatiquement. Les dépendances d'un paquet installées en même temps que lui entrent dans cette catégorie. Ainsi, Aptitude garde une trace du type d'installation opéré. Si plus aucun paquet n'est présent pour justifier la présence d'un ou plusieurs paquets marqués `M`, ceux-ci sont désinstallés. Voilà qui est séduisant pour un système classique, beaucoup moins pour une machine de développeur où la désinstallation de certaines bibliothèques n'est sans doute pas souhaitée. Bien entendu, il est possible de forcer la main au système en utilisant une option spécifique ou encore d'installer explicitement la bibliothèque désirée. Cela revient à agir en fonction du fonctionnement de l'outil et non plus l'inverse.

Une autre fonctionnalité offerte est la rétention de paquet. Il est possible d'interdire la mise à jour d'un ou plusieurs paquets vers une version donnée ou même de bloquer toute modification. Ce type de fonctionnalité peut être utile si l'on souhaite gérer des problèmes de compatibilité avec des éléments logiciels particuliers.

Conclusion

Aptitude semble être le remplaçant idéal pour le vieux `dselect`. Comme dit précédemment, l'utilisateur lambda préfèrera sans doute une application graphique correctement intégrée à son *desktop* (KDE ou GNOME). L'interface Curses est déroutante et rappelle vraiment celle de `dselect`. Elle ne séduira sans doute pas l'utilisateur avancé, préférant la souplesse et la clarté de la ligne de commande. C'est donc là qu'Aptitude finalement trouve sa place grâce à ses fonctionnalités de marquage des dépendances, évitant ainsi de devoir utiliser des outils comme `deborphan`.

LIENS

- **La page d'information du paquet Aptitude :** http://packages.debian.org/testing/admin/aptitude
- **Aptitude sur Wikipédia :** http://en.wikipedia.org/wiki/Aptitude
- **Aptitude sur le Wiki Debian** http://wiki.debian.org/Aptitude?action=show&redirect=aptitude

Denis Bodor,

db@ed-diamond.com
lefinnois@lefinnois.net

Éric Lacombe, Matthieu Barthélemy

Les nouveautés du noyau 2.6.20

De retour pour cette nouvelle édition, nous vous proposons un Corner consacré au noyau Linux 2.6.20, qui devrait être sorti au moment où vous lirez ces lignes. Linus a souhaité que la 2.6.20 soit une version de stabilité. Ainsi, il n'hésite pas à retirer (revert) des patchs de fonctionnalité, pourtant inclus dans la -rc1, qui posent problème et préfère les réintégrer dans le cycle suivant, celui du 2.6.21. Adrian Bunk, quant à lui, continue son travail sur la stabilisation du 2.6.16 qui en est à la 38ème révision, et maintient une liste de régression entre le 2.6.16 et la dernière version afin que la fiabilité de la série 2.6 soit progressivement éprouvée. Cependant, il y a de quoi assouvir votre curiosité avec l'arrivée de cette nouvelle mouture qui inclut quelques belles surprises. Nous terminons ce Kernel Corner sur l'explication du bug le plus monopolisant pour la communauté des développeurs du noyau Linux. Sa résolution a pris tout le mois de décembre !

➤ [ACTUALITÉ] LE NOYAU 2.6.20

Éric Lacombe, tuxiko@free.fr

La virtualisation

Avant d'aborder le sujet de la virtualisation du noyau, notons l'introduction du support pour la compilation optimisée pour le processeur Intel Core 2 Duo (processeur pouvant assister matériellement la virtualisation). Pour l'instant, GCC ne gère pas encore l'optimisation possible avec cette architecture (prévue dans la version 4.3). Ainsi, le support est ajouté principalement pour prendre en compte les lignes de cache de 64 bits du Core 2 Duo.

Pour commencer réellement sur la virtualisation, notons l'intégration de KVM (*Kernel-based Virtual Machine driver* créé par Avi Kivity) dans cette version du noyau. Nous en avions parlé dans le Kernel Corner 90 assez longuement. Cependant, une activité débordante s'est déroulée depuis et se déroule encore au sein de KVM. Nous allons y revenir, mais notons déjà la réimplémentation des *shadow page tables*, point critique pour les performances. Nous trouvons également en vedette une fonctionnalité de taille, introduite par Ingo Molnar, qui ajoute le support de la paravirtualisation pour KVM ! Un travail de titan est à l'œuvre afin de doter Linux de fonctionnalités évoluées au niveau de la virtualisation.

Nous avions mentionné dans le KVM précédent que l'implémentation des shadow page tables était basique et loin d'être optimale en termes de performance. Maintenant, ces tables sont mises en cache afin d'éviter les reconstructions complètes des tables de pages des invités à chaque changement de contexte. Malgré un certain nombre de difficultés à surmonter, les systèmes invités tournant sur KVM sont plus rapides et réactifs.

Terminons avec l'introduction de la paravirtualisation dans KVM, qui n'est pour l'instant implémentée que pour des systèmes Linux sur Linux. En plus de cela, Ingo Molnar ajoute le support de la fonctionnalité du cache matériel du cr3 (cr3 est le registre de contrôle numéro trois qui contient notamment l'adresse de la PGD – *Page Global Directory* – du processus s'exécutant) des processeurs Intel de type VT-x, allant une étape au-dessus de l'implémentation du cache logiciel des shadow page tables pour les architectures Intel. Les gains lors de l'utilisation de la paravirtualisation et du cache matériel du cr3 en ce qui concerne les changements de contexte et les *flushs* du TLB (*Translation Lookaside Buffer*) sont très impressionnants. Dans cette configuration, KVM est relativement proche d'un système sans virtualisation, en termes de performance sur les aspects mentionnés. Notons que si le système Linux invité intégrant KVM avec la paravirtualisation ne détecte pas les capacités de paravirtualisation du côté de l'hyperviseur, il passe automatiquement au mode de fonctionnement de virtualisation complète. Pour plus de renseignement au sujet de KVM, jetez un œil à **http://kvm.sourceforge.net**.

Afin de supporter différents types de virtualisation, Rusty Russel a intégré la structure `paravirt_ops`, contenant l'ensemble des appels critiques qu'un hyperviseur doit implémenter. Dans le cas où `CONFIG_PARAVIRT` est activé, le fichier d'en-tête `paravirt.h` est inséré dans le code afin que toutes les opérations critiques qui ont besoin d'être remplacées par des appels à l'hyperviseur, le soient.

➤ [ACTUALITÉ] LE NOYAU 2.6.20

Matthieu Barthélemy, bonsouere@gmail.com

Les périphériques

La version 2.6.19 ayant vu apparaître la découverte parallélisée des périphériques PCI, ce sont cette fois les périphériques USB et SCSI qui peuvent en bénéficier. Plus précisément, la découverte de périphériques sur un bus SCSI peut être asynchrone : les périphériques s'initialisent tandis que le système effectue en parallèle d'autres opérations de *boot*, ceci étant possible simultanément sur plusieurs bus SCSI. Concernant l'USB, l'initialisation de chaque périphérique par son driver peut être effectuée dans un *thread* noyau (kthread). Une légère amélioration du temps de boot peut être observée sur un système monoprocesseur, l'effet le plus marqué se fera apprécier sur les systèmes multicœurs/multiprocesseurs. Pour tester, il faudra respectivement activer les options `SCSI_SCAN_ASYNC` (*SCSI device support -> Asynchronous SCSI scanning*) et/ou `USB_MULTITHREAD_PROBE` (*USB Support -> USB Multi-threaded probe*). Cependant, coup de théâtre, cette fonctionnalité aura été de courte durée pour le bus PCI : elle sera marquée *broken* pour le 2.6.20 et devrait être retirée pour le 2.6.21. En effet, par définition il n'y a pas de support d'opérations asynchrones au niveau du bus PCI, alors que c'est le cas pour l'USB et SCSI. Linus Torvalds et Andrew Morton ont invoqué cette raison, plus le fait que beaucoup de drivers ne fonctionnent plus avec cette option activée.

Ce travail effectué sur le noyau trouve un complément dans les nouveaux projets d'init rapide et/ou parallélisé, tels que InitNG, UpStart... Une fois ceux-ci généralisés dans les distributions, nous ne devrions plus avoir à rougir en démarrant notre distribution favorite face à ses concurrents moins libres.

Toujours au sujet des bus, le développeur Benjamin Herrenschmidt introduit un ensemble de nouvelles structures apportant la « notification des événements de bus », se déclenchant à quatre occasions : ajout et retrait de périphérique, attribution et désattribution d'un pilote pour un périphérique. Un exemple dans lequel ce nouveau code trouve son utilité est le cas où un pilote est dépendant de la présence d'un autre pilote et/ou périphérique, et doit donc savoir quand ceux-ci apparaissent dans le système.

L'architecture interne du noyau se voit remaniée en ce qui concerne les périphériques d'interaction avec l'utilisateur (dits « HID », pour *Human Input Device*), tels que clavier, souris, manette... Le code utilisé jusqu'à maintenant par les périphériques de type USB (module noyau `usbhid`) a été adapté en une couche générique pouvant être utilisée par les autres catégories : Bluetooth, PS2... Elle assure principalement le dialogue entre le système et les périphériques, et la remontée d'événements déclenchés par ces périphériques.

Le travail sur la libATA générique continue, avec des bogues résolus, le support de nouveaux *chipsets* (Marvel, Winbond...) et de fonctionnalités supplémentaires sur l'existant (NCQ ou *Native Command Queuing*, cf. Kernel Corner 86, pour les puces SATA Nvidia). Côté système de fichiers, très peu de changements à noter. Utile pour les administrateurs système, des statistiques sur la consommation d'opérations d'entrées/sorties par processus font leur apparition dans `/proc`. Il est ainsi possible de savoir simplement les volumes de lecture et écriture effectués par un processus via un `cat` sur `/proc/<le pid>/io`. Ces informations sont cumulatives. Pour avoir les chiffres instantanés, il vous faudra donc faire la différence entre la valeur actuelle et une valeur prise peu de temps auparavant. N'oublions pas que depuis le 2.6.17, Linux nous permet de modifier la priorité d'accès aux entrées/sorties d'un processus, ce qui nous permet donc d'effectuer un `ionice` sur un processus gourmand en accès disques qui en gênerait d'autres.

Dégradées depuis les débuts du noyau 2.6, les écritures en mode Direct IO (`O_DIRECT`) sur les périphériques de type bloc (disques) voient leurs performances significativement améliorées ; ceci est valable pour, par exemple, les SGBD, qui préfèrent accéder directement au périphérique (via `/dev/{s|d}X`) sans passer par le cache du noyau. Ce progrès a pu être obtenu en simplifiant le code exécuté dans le cas d'un transfert vers un périphérique de bloc.

Les systèmes de fichiers

Le système de fichiers OCFS2 peut désormais gérer le `atime` relatif, c'est-à-dire que la métadonnée de chaque fichier indiquant sa date de dernier accès peut n'être modifiée que si elle est plus ancienne que le `ctime` (date de création) ou le `mtime` (date de modification) au lieu de l'être systématiquement. Ceci améliore les performances et est similaire à l'option `noatime` (ne pas modifier la valeur `atime` quand le fichier est accédé en lecture).

Le réseau

La partie réseau de Linux bénéficie aussi d'évolutions, quelques-unes sont à signaler en particulier. Tout d'abord, le support du NAT (*Network Address Translation* ou translation d'adresse) fait son apparition dans le module `nf_conntrack`, couche de *connection tracking* (suivi de connection) générique de Netfilter gérant l'IPv4 et IPv6. Ce module est voué à remplacer `ip_conntrack` à brève échéance (les développeurs

[ACTUALITÉ] LE NOYAU 2.6.20

Matthieu Barthélemy, bonsouere@gmail.com

Netfilter parlent de son retrait dès Linux 2.6.22). Plusieurs *NAT helpers* qui n'existaient auparavant que pour `ip_conntrack` font leur apparition, en particulier pour les protocoles FTP, IRC, PPTP, SNMP, SIP et TFTP. Il s'agit de petits modules dédiés à la gestion de paquets spécifiques à un protocole de couche applicative.

Un protocole de plus est maintenant géré. Il s'agit d'UDP-lite, un protocole dérivé d'UDP. Comparé à ce dernier qui rejette un paquet entier si sa somme de contrôle est incorrecte, UDP-lite autorise un paquet incorrect à être traité, laissant à la couche applicative le soin de décider s'il est exploitable ; moins d'informations perdues peut signifier des améliorations dans la transmission de flux « en direct », tels que son ou vidéo.

Le monde des pilotes de périphériques ne connaît pas non plus de bouleversements majeurs avec cette nouvelle version, mais on y distingue :

- du côté des webcams, l'ajout du module `usbvision` apporte la gestion par Linux d'une cinquantaine de modèles USB supplémentaires (puces Nogatech), tandis que le nouveau module apporte de son côté la gestion des puces de type OV76xx ;
- des améliorations dans la prise en charge des puces Intel i915 par le DRM (*Direct Rendering Manager*) ;
- le support de l'interface Ethernet Atmel présente dans les microcontrôleurs AVR32 (Cf. Kernel Corner 89) ;
- la gestion des tuners DVB USB Pinnacle 400e ;
- la gestion de l'Apple Motion Sensor ;
- l'apparition de pilotes pour les cartes Ethernet de type NetXen 1-10G et Chelsio ;
- la poursuite du retrait des pilotes de périphériques de son utilisant l'obsolète sous-système OSS.

[ACTUALITÉ] LE NOYAU 2.6.20

Éric Lacombe, tuxiko@free.fr

Les travaux différés

Les `Workqueues` sont un mécanisme intégré depuis le développement de la 2.6. Ils sont mis à disposition des threads noyau afin qu'ils puissent soumettre des travaux qui seront exécutés de façon différée dans un contexte de processus. En effet, ce travail est exécuté soit par l'*idle thread*, soit par un thread spécialisé associé à la `workqueue`. Ainsi, les fonctions différées peuvent être bloquantes. Notons la différence avec les `Tasklets`, qui sont, quant à elles, exécutées dans un contexte d'interruption et ne peuvent donc pas bloquer. Dernière remarque, les `workqueues` contiennent toutes une liste de travaux, lesquels sont représentés par des structures `work_struct`.

David Howells a remanié les `work_struct`, afin d'éviter des gaspillages de mémoire. En effet, jusqu'à présent le descripteur de travail `work_struct` embarquait toujours un *timer* logiciel, lequel pouvait être utilisé pour retarder le travail à effectuer. Cependant pour plus de la moitié du code du noyau, ce timer n'est pas utilisé, occupant ainsi de la place en mémoire pour rien. Le changement principal est d'avoir enlevé le timer de la `work_struct`, créant ainsi un descripteur primitif. Dans le cas où l'on souhaite utiliser le timer, une nouvelle structure regroupant un timer et une `work_struct` est utilisée. Ainsi, on obtient des gains significatifs sur la taille des `work_struct` :

	Mots Mémoire	Archi 32 bits	Archi 64 bits
Tel quel	12	48 octets	96 octets
Non temporisable	4	16 octets	32 octets
Temporisable	10	40 octets	80 octets

Notons également la diminution en taille dans le cas des `work_struct` temporisables. Cela est dû au retrait du pointeur (= un mot mémoire) `data` de la `work_struct`. La fonction de travail prenait jusqu'à présent ce pointeur en argument. Maintenant, pour que la fonction de traitement puisse accéder à ces données, il suffit d'embarquer la `work_struct` au sein d'une structure privée et ensuite d'utiliser la primitive `container_of` (Cf. Kernel Corner 86).

Les timers

Au niveau de la fréquence du timer, la valeur par défaut a été positionnée à 300 Hz (à la place de 250 Hz dans les versions précédentes), afin d'optimiser le traitement multimédia. En effet, le taux de trames par seconde utilisé en Europe est de 25, mais pour les États-Unis de 30. 300 étant un multiple à la fois de 25 et de 30, il a été jugé convenable de faire cette modification (proposée par Alan Cox). De plus, 300 se trouve être également un multiple de 50 et de 60, correspondant aux taux de trames par seconde en travail *interlacé* dans les deux cas considérés

➤ [ACTUALITÉ] LE NOYAU 2.6.20

Éric Lacombe, tuxiko@free.fr

précédemment. Finalement, les performances induites par ce changement dans les autres types d'activité du noyau sont similaires à celles d'avant.

Dans le domaine des timers, une nouveauté importante est l'ajout par Arjan van de Ven des fonctions `round_jiffies` et `round_jiffies_relative`. Tout d'abord, expliquons brièvement le rôle de la variable `jiffies` présente dans le noyau. Elle est incrémentée à chaque interruption – *tick* – du timer, permettant la création de timers logiciels dont l'échéance peut être donnée relativement à cette variable. Les fonctions introduites arrondissent une valeur de `jiffies` à la seconde supérieure. Le principal objectif de ces fonctions est de regrouper sur le même *jiffy* les timers logiciels qui n'ont pas de contraintes fortes dans le temps. Ainsi, on évite l'enclenchement de nombreux timers à des dates différentes dans la même seconde. De plus, avec l'utilisation du patch *dynamic ticks* (Cf. Kernel Corner 87), les états de profond sommeil de la CPU durent plus longtemps. Cela introduit par conséquent un gain d'énergie significatif. Notons que pour les architectures à plusieurs CPU, le moment de réveil exact des différents CPU est modifié en fonction de leur position, cela afin d'éviter qu'il puisse se réveiller exactement au même moment et toucher les mêmes lignes de cache par exemple (qui introduirait des aller-retours supplémentaires non négligeables).

Un casse-tête pour résoudre un bug de corruption de fichiers

Durant le mois de décembre, une bonne partie des développeurs de plus haut niveau du noyau Linux était occupée à traquer un *bug* de corruption de fichiers soumis par un utilisateur et affectant le 2.6.19.1 Après de longs efforts, de nombreux tests, un audit minutieux des sous-systèmes suspectés, Linus a pu mettre au point un programme capable de déclencher facilement la corruption. Après avoir élagué les différentes possibilités, Linus a proposé le soir de Noël un patch permettant de corriger le problème.

[ACTUALITÉ] LE NOYAU 2.6.20

Éric Lacombe, tuxiko@free.fr

La corruption de fichier pouvait se produire dans une situation de *mapping* (i. e. mise en correspondance) partagé de fichier (*shared mmap*). Le bug était une *race condition*. Afin d'appréhender correctement le bug, expliquons brièvement le fonctionnement du *page writeback* (i. e. le traitement des pages mémoires nécessitant une réécriture sur le disque).

Une page de mémoire physique peut être mappée dans l'espace d'adressage de différentes applications (shared mmap). Pour chacune d'elle, une entrée dans une table de page (PTE : *page table entry*) de l'application est créée. Elle établit la correspondance entre l'adresse physique de la page et l'adresse virtuelle de l'espace d'adressage du programme y faisant référence. Pour chaque PTE, les 12 bits de poids faibles sont utilisés comme bits de contrôle (seulement les 20 bits de poids forts sont nécessaires pour adresser une page de 4 Ko). L'un d'eux est le *dirty bit* (bit attestant de la modification d'une page) et le processeur le positionne à 1 lorsque la page est modifiée par l'application. Dans la situation où plusieurs PTE pointent sur la même page physique, des différences d'état du dirty bit sont courantes. Ainsi, il est nécessaire de mettre en place un moyen de traque de toutes ces PTE. Le noyau Linux maintient un ensemble de descripteurs de pages correspondant exactement au nombre de pages disponibles dans la mémoire physique. Ces descripteurs contiennent différents champs dont un *member flag* statuant entre autres sur l'état dirty de la page. L'ensemble des PTE associées à la page physique sont également accessibles au travers de ce descripteur, tout comme les blocs correspondants sur le disque dur, s'il s'agit d'un mapping de fichier.

Venons-en à l'unité d'écriture sur un disque : le bloc dont la taille fait 512 octets. Si une page mémoire mappe une partie d'un fichier, elle peut en contenir 8 blocs. Ces blocs sont représentés dans le noyau par des descripteurs appelés *buffer heads*. Leur utilisation est maintenant très réduite, car un cache de page est utilisé globalement dans le noyau. Ainsi, ces buffer heads sont un reste datant du 2.4, toujours employé par certains systèmes de fichiers, notamment ext3 (ainsi qu'ext4 actuellement), afin d'effectuer les écritures de page mémoire sur le disque. Ces buffer heads ne sont créées qu'au besoin par le système de fichiers. Ainsi, seuls les blocs dirty sont représentés avant écriture sur le disque. Vous l'aurez deviné, ces buffer heads contiennent également un dirty flag.

Venons-en au problème. Lorsque le gestionnaire de mémoire s'aperçoit qu'une page a été modifiée via la lecture du dirty bit d'une PTE ou une opération explicite d'une application, il va appeler la fonction `set_page_dirty`. Cette dernière appelle la fonction de rappel définie par le système de fichiers, si elle existe. Cette dernière n'est quasiment jamais définie. Aussi, le gestionnaire de mémoire, dans cette situation courante, traverse la liste de buffer heads et les marque dirty afin que le système de fichier soit au courant de l'état des blocs et les recopie sur le disque (via sa méthode `writepage`).

Le problème apparaît lorsqu'une même page physique est partagée. À partir de ce moment, deux traitements simultanés peuvent se produire. Une écriture disque induite par le système de fichier via `writepage` peut s'effectuer alors que le gestionnaire de mémoire est en train de parcourir la liste des buffer heads pour les marquer dirty. A la fin des opérations d'E/S, le système de fichiers va positionner à 0 le dirty flag des buffer heads, car il vient tout juste d'écrire les blocs correspondants sur le disque. Cependant, le gestionnaire de mémoire les avait positionnés à 1. Ainsi, des blocs de données en mémoire nécessitant d'être répercutés sur le disque ne le seront pas. Cette race condition entraîne ainsi la corruption du fichier.

Le patch temporaire proposé par Linus correspond en gros à effectuer un nouvel appel à `set_page_dirty`, dans la fonction appelée par le système de fichier lorsqu'il entame sa remise à zéro du dirty flag des buffer heads.

Concluons sur le fait que Linus s'insurge contre l'utilisation « de nos jours » des buffer heads dans les systèmes de fichiers comme ext3, car ils passent outre le page cache, et induisent des problèmes de performance significatifs, notamment lors de l'exécution de `readdir`. Pour l'instant, ext4 n'apporte pas de modification à ce niveau, mais Theodore Tso a proposé, en réponse à Linus, de corriger cela.

Finissons cette section « bug » en mentionnant qu'il est désormais possible, pour l'architecture i386, de charger le noyau à n'importe qu'elle adresse alignée sur un multiple de 4 Ko et en dessous de 1 Go. Cette fonctionnalité est intéressante pour ceux qui testent des noyaux afin de générer des *crash dumps* (Cf. le sous-système Kexec).

Denis Bodor

FOSDEM : dernière ligne droite

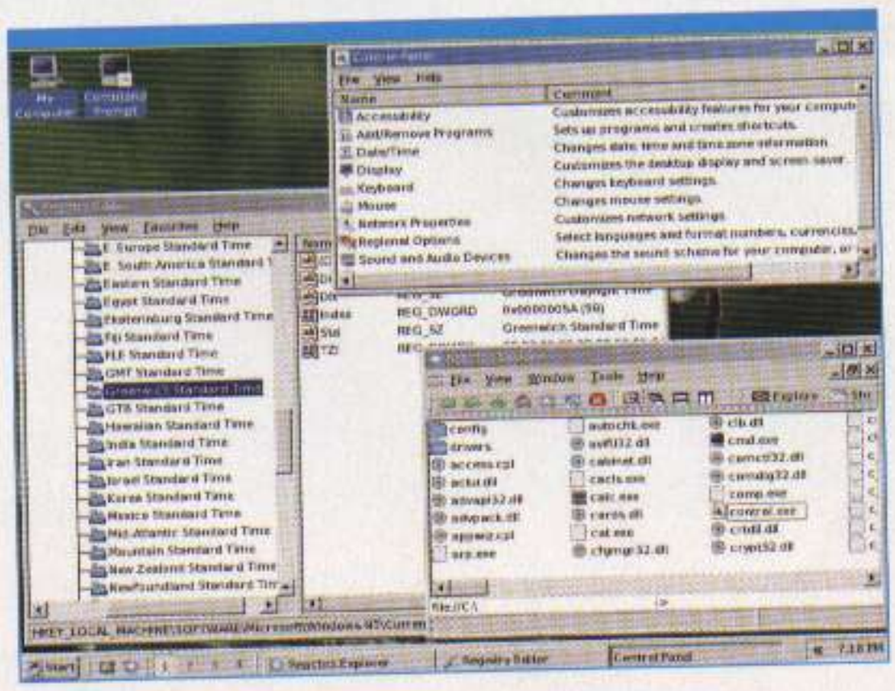
Nous y sommes, fin du mois aura lieu la rencontre européenne des développeurs Open Source à Bruxelles. A quelques semaines de l'évènement, les informations se précisent et on ne peut qu'en conclure que cette édition risque d'être exceptionnelle.

Que des huiles !

Première bonne nouvelle, même si on s'y attendait, Andrew Morton sera bel et bien présent cette année. Rappelons qu'il est le responsable de la branche 2.6 du noyau Linux, qu'il a donc la grande responsabilité de décider quel code ira dans quelle version du noyau. Andrew tiendra une conférence le samedi 24 salle Janson à 16 heures. Si nous ne savez pas qui est ce monsieur (honte sur vous !), visitez sa page personnelle **http://www.zip.com.au/~akpm/** ou, plus simplement, sa page sur Wikipédia : **http://en.wikipedia.org/wiki/Andrew_Morton_%28computer_programmer%29**.

Puisque nous parlons des prestigieux conférenciers de l'édition 2007, enchaînons avec quelques autres « pointures ». Nous avons tout d'abord le sympathique et superactif Miguel De Icaza qui présentera le projet Mono (salle Chavanne, samedi 14h), mais aussi Ronald G. Minnich du projet LinuxBIOS (salle Janson, samedi 15h), Jim Gettys du projet « *One Laptop Per Child* » (salle Janson, samedi 11h) ou encore Jeremy Alisson, développeur du projet Samba (Salle Janson, samedi 16h).

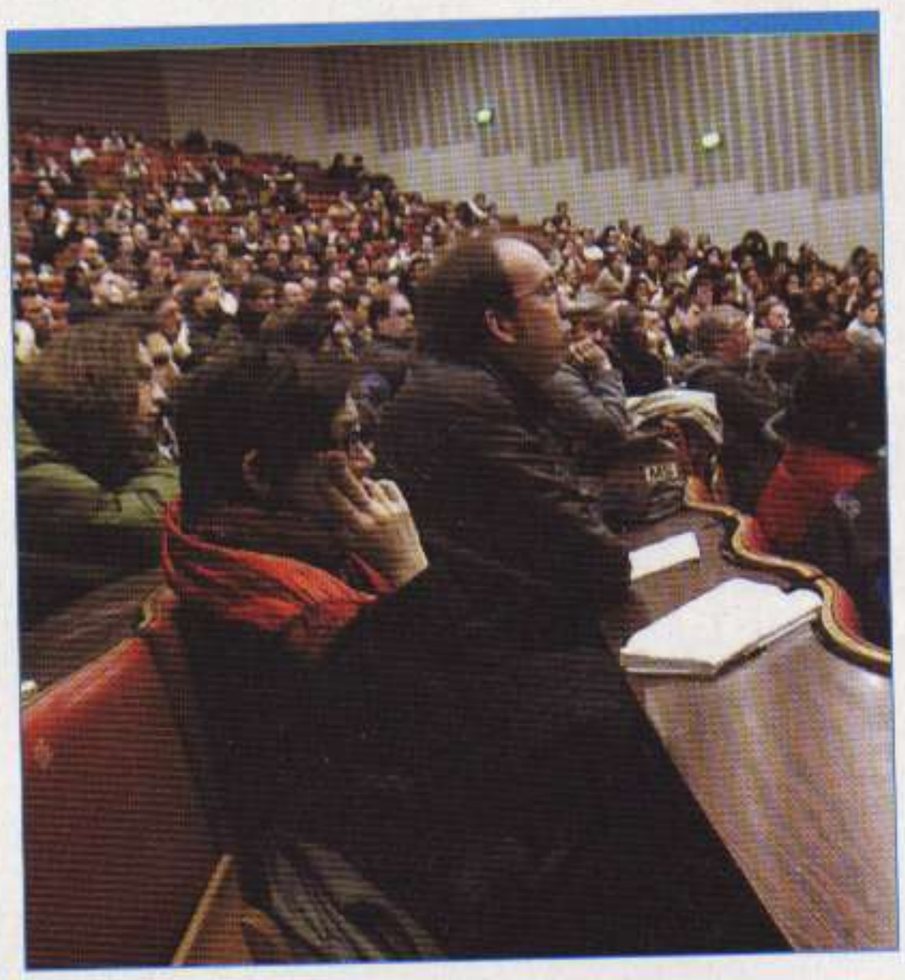
À noter, également, la présence d'une conférence sur ReactOS donnée par Aleksey Bragin, coordinateur du projet. ReactOS est un système d'exploitation ayant pour but de fournir une compatibilité binaire avec Microsoft Windows XP et NT. En d'autres termes, il s'agit d'une tentative d'implémentation en Logiciel libre d'un système compatible Windows. Comme le montre la capture sur cette page, le projet semble relativement avancé. Cette conférence sera l'occasion de connaître les motivations profondes derrière le développement ainsi que les difficultés rencontrées.



Archives

Si vous n'avez pas la chance de pouvoir vous rendre à Bruxelles cette année pour assister aux conférences, sachez que, comme les années précédentes, les archives vidéo vous seront bien utiles. Remarquez que si vous êtes visiteur du FOSDEM 2007, ces archives vous concernent tout autant. Difficile en effet d'être dans plusieurs salles de conférence en même temps.

Le site officiel, **http://fosdem.org/** propose une section spéciale où vous pourrez trouver des vidéos, des images et les *slides* de la quasi-totalité des présentations et conférences. Pour les vidéos, un miroir a été mis en place : **http://ftp.belnet.be/mirrors/FOSDEM/**.



Denis Bodor,

db@ed-diamond.com :: lefinnois@lefinnois.net

Sébastien Aperghis-Tramoni

Les Journées Perl 2006

Revenant cette année à Paris, les Journées Perl 2006 se déroulent à nouveau à la Cité des Sciences et de l'Industrie de la Villette, dans l'espace consacré aux nouvelles technologies.

Un succès malgré tout

La Cité des Sciences

Si on peut reprocher une chose aux organisateurs, c'est bien le manque de transparence et de communication. Initialement prévue plus tôt dans l'année, elle a finalement été repoussée jusqu'à fin novembre, son organisation se faisant cahin-caha, mais dans une grande opacité. Elle a malgré tout réuni plus de 70 personnes, qui ont pu choisir parmi une trentaine de présentations différentes.

Premier jour, samedi 25 novembre

Ce matin du premier jour débute, comme il se doit, par l'inscription de chaque participant. Chacun reçoit la désormais traditionnelle sacoche en tissu à l'effigie du logo officiel des Mongueurs, contenant cette année un exemplaire du dossier de *Linux Magazine France* consacré à Perl, un bloc-notes, un stylo et des brochures Talend, ainsi que des brochures et *goodies* RedHat. Les t-shirts de la conférence, eux aussi avec le logo des Mongueurs, sont de couleur beige pâle pour les participants et orange pour les organisateurs (à croire qu'un Léon Brocard est passé par-là...)

Une partie du public des Journées Perl 2006

Sans surprise, c'est avec du retard que David Elbaz ouvre la conférence, et invite chaque participant à se présenter. Il remercie par avance les personnes qui se sont déplacées pour venir assister aux nombreuses présentations proposées.

Laurent Gautrot débute en montrant comment installer sa propre version de Perl sur Nokia 770, une tablette internet qui tourne sous Debian GNU/Linux (cf. GLMF n°81). Comme l'installation normale de Perl est trop grosse à son goût, il lui fait subir une cure d'amaigrissement assez radicale.

Puis, vient mon tour et j'explique comment écrire du code Perl relativement moderne, mais compatible avec les anciennes versions du langage, par l'utilisation judicieuse de modules et éléments de syntaxe compatibles.

Xavier Caron enchaîne en montrant comme améliorer la qualité de son code Perl. Propagande CPAN-iste, protocole TAP, couverture de code et couverture fonctionnelle.. Xavier semble vouloir tout expliquer, et je me demande ce qu'il me restera à dire ! Surtout qu'il brosse rapidement, mais doit certainement allécher plus d'un programmeur ;-)

Le déjeuner consiste en un couscous géant dans un restaurant pas loin de la Cité des Sciences. L'après-midi va être somnol^Wstudieuse !

Après ce consistant repas, Olivier Mengué commence par une courte présentation des notions de base sur les ORM (*Object Relational Mapper*), ces briques logicielles qui permettent de piloter une base de données comme une collection d'objets, évitant ainsi l'écriture de code SQL. Olivier enchaîne ensuite sur l'ORM de Django (Python), et est suivi de Noël Rocher qui parle d'Hibernate (Java), puis de Nicolas Chuche

qui aborde ActiveRecord (Ruby) et enfin de Laurent Gautrot qui présente `Class::DBI` (Perl !). La plupart utilisent comme exemple la base de données d'une association de randonneurs, autorisant ainsi le public à facilement comparer la mise en application d'une même base avec ces différents ORM et langages.

Le dîner au restaurant La Pouchla

Pendant ce temps, dans l'autre salle, Arnaud Assad montre tout l'intérêt qu'il y a à utiliser QPSTMPD, le `mod_perl` du mail. Il rappelle entre autres que bien qu'il soit écrit en Perl, ses performances n'ont rien à envier aux autres logiciels existants, puisqu'il gère plus de 2 millions de mails par jour sur `apache.org`, et un nombre guère moins conséquent sur `perl.org`.

Michaël Scherer parle de manière curieuse de Djabberd, un serveur Jabber écrit en Perl, en illustrant avec des photos incongrues.

Ingrid Falk, chercheuse au CNRS, explique ensuite comment elle utilise Perl pour générer un lexique syntaxique, utilisant pour ce faire plusieurs modules Perl (`XML::LibXML`, `Parse::RecDescent`, `Graph`). Le but étant d'automatiser la production du lexique LDAL du projet SynLex, dans le cadre des systèmes de traitement automatique du langage (TAL).

Suit Thierry Hamon qui présente le projet ALVIS, un moteur de recherche spécialisé dans l'intégration d'informations sémantiques par une analyse linguistique des documents au travers d'une plate-forme exploitant des outils de TAL.

De retour dans la salle `undef`, on continue sur le sujet d'ORM avec une présentation de `DBIx::DataModel` par Richard Gerber, Laurent Dami n'ayant pu venir à Paris. Ce module est encore un autre ORM, mais avec une vision orientée UML de la base de données. Nicolas Rennert termine sur ce sujet avec son ORM à lui, `PerlPersistentObject`.

Viennent alors deux présentations sur les environnements web avancés en Perl, avec Damien Krotkine qui introduit Mason de manière amusante, et Erik Colson qui montre les bases de Catalyst.

La soirée se passe au restaurant *La Pouchla*, découvert pendant les réunions alternatives de **Paris.pm**.

Second jour, dimanche 26 novembre

Pour ouvrir cette seconde journée, c'est Christian Aperghis-Tramoni qui commence salle `undef` par parler de Parrot, la machine virtuelle qui servira à exécuter le bytecode Perl 6. Il montre tout l'intérêt que cette machine virtuelle offre par son assembleur bien plus propre que les assembleurs machine (et en particulier de l'infâme dialecte Intel x86).

Suit Stéphane Payrard qui reprend une présentation d'Audrey Tang à propos de Pugs, de son évolution et de celle qu'il a induite dans les avancées simultanées de Perl 6 et Perl 5, comme le projet *6-on-5* qui consiste à traduire le code Perl 6 en Perl 5 afin de pouvoir l'exécuter avec l'interpréteur Perl5 actuel.

Il est suivi de Jonathan Worthington, un programmeur Parrot britannique, qui présente (en anglais) la théorie derrière les langages de programmation. Comment un code source est analysé et validé au travers d'une grammaire (qui définit le langage utilisé), converti en un arbre de syntaxe abstrait (AST, *Abstract Syntax Tree*) et enfin exécuté ou converti en assembleur ou bytecode pour machine virtuelle.

Stéphane Payrard et Jonathan Worthington

Pendant ce temps, salle Talend, David Morel essaye (un peu laborieusement de son propre aveu) de montrer toute la puissance et la richesse que Catalyst met à disposition de ses utilisateurs par la très grande quantité de modules d'extension disponibles sur le CPAN.

J'enchaîne avec une présentation sur les conséquences de traîner sur IRC, ce lieu de perdition moderne. Par exemple, les *bots* comme `purl`, `babel`, `meta`, `assemble`. Par exemple, les défis comme le code pour mettre à jour la liste de Phalanx 100. Par exemple, les modules du *core* que certains *porters* vous proposent *gentiment* d'en prendre la maintenance. Enfin, des défis plus effrayants comme l'écriture de `Lingua::FR::Inflect`.

En sortant de la salle, votre serviteur a la surprise de découvrir qu'une thésarde en sociologie était venue pour assister à la conférence, et semble-t-il plus spécifiquement à cette présentation afin d'étudier la socialisation des *geeks*. L'ayant raté, je lui montre les *slides* et une discussion s'engage entre elle et quelques conférenciers sur ce sujet. Nous ignorons si elle a obtenu les informations qu'elle pensait trouver ;-)

Après la pause déjeuner, Stéphane Payrard commence en parlant des règles, les remplaçantes des expressions régulières en Perl 6, et d'une de leurs (nombreuses) applications, le filtrage par motif.

J'enchaîne avec une présentation basique sur les tests qui reprend certains des points abordés dans l'article paru dans le GLMF n°88.

David Landgren explique l'algorithme Aho-Corasick

Pendant ce temps, salle Talend, Philippe Bruhat expose son nouveau module, `Net::Proxy`, qui propose certaines fonctionnalités amusantes comme le support de deux services (HTTPS et SSH) sur un même port TCP.

Suit Richard Gerber qui passe là aussi une présentation que Laurent Dami avait faite à *YAPC::Europe* 2006 à Birmingham, cette fois sur la publication sur le net des décisions de la justice genevoise.

Retour en salle `undef` pour la dernière présentation de la conférence par David Landgren qui explique les nouveautés de Perl 5.10. Suivant le sujet d'assez près, il est intarissable et parlera pendant plus de 50 minutes sur l'opérateur *defined-or* `//`, les variables d'état `state $var`, le *smart match* `~~` pour comparer des éléments complexes, le moteur d'expression régulière qui a été *dérécursé* par Dave Mitchell et étendu avec de nouvelles fonctionnalités par Yves Orton. Sans compter les gains substantiels en termes de mémoire, d'occupation disque et de rapidité.

Vient l'heure des présentations éclair, sous l'œil des gros bras de la conférence, Arnaud Dupuis et Emmanuel Seyman ;-)

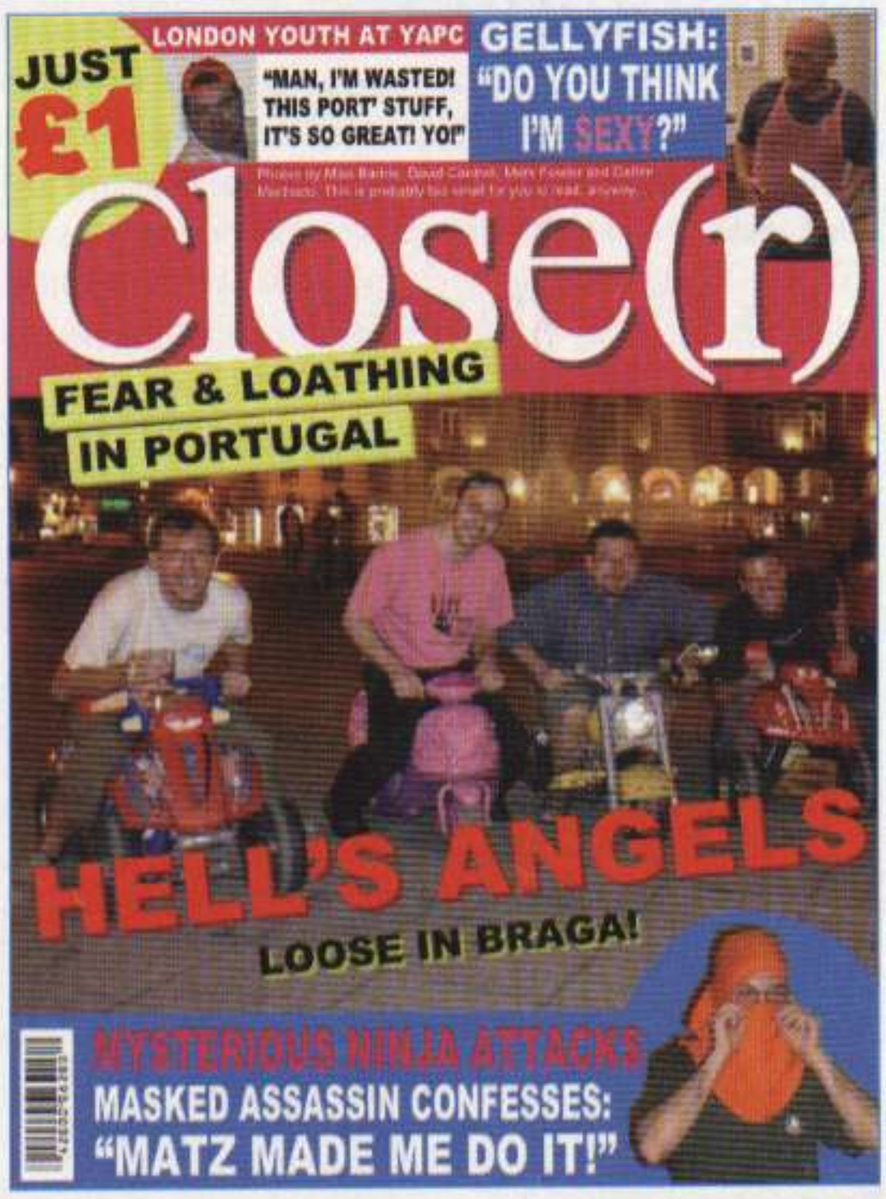
Philippe Bruhat commence en parlant de son module `Acme::MetaSyntactic` sur fond de musique de la série Batman. Olivier Thauvin enchaîne sur `VFSSimple`, une couche d'abstraction d'accès aux fichiers. Puis j'expose de manière très directe les Règles de `Sys::Syslog`. Vient ensuite Christian Aperghis-Tramoni qui montre l'enseignement Perl qu'il donne dans différents endroits du globe, et les groupes de *mongers* qui commencent à s'y créer. Suit la présentation de Talend, l'un des sponsors, qui embauche des développeurs Perl et Java.

Puis David Landgren parle (trop) rapidement de `Regexp::Assemble` (voir la Perle de Mongueurs de ce numéro pour une présentation de ce module). Jonathan Worthington propose ensuite l'organisation d'un *hackathon* Perl européen, sur le modèle de celui qui s'est tenu à Chicago, USA. Paul Gaborit monte alors pour parler de son bien utile projet de traduction de la documentation de Perl.

Vient alors David Elbaz qui résume l'activité du groupe de travail « Articles des Mongueurs » de Perl, et indique qu'il me passe la main comme nouveau coordinateur. Suit la présentation de RedHat, un autre sponsor, qui embauche aussi. Puis Stéphane Payrard explique la « thématisation », un mécanisme que Perl 6 a emprunté au japonais et utilise des exemples prévus pour m'embarrasser ;-)

Jean Forget enchaîne en expliquant de manière amusante en quoi c'est une bonne chose de réinventer la roue. Puis, David Landgren revient pour parler cette fois-ci des résumés de la liste Perl5 Porters dont il assume actuellement la rédaction. Enfin, Philippe Bruhat termine avec sa présentation sur la « vérité » des faits qui se sont produits à Braga lors de *YAPC::Europe* 2005.

Le magazine parodique des Perl de Mongueurs :



David Elbaz remercie les conférenciers qui ont fourni autant de présentations, et sur des sujets aussi divers et intéressants, le public qui est venu assez nombreux malgré la publicité qui ne fut pas aussi importante que d'autres fois et, sur ces mots, termine les *Journées Perl* 2006. Bien que cela ne soit pas officiellement confirmé, les milieux autorisés laissent entendre que les *Journées Perl* 2007 se dérouleront peut-être à Lyon.

Bilan

Au final, cette conférence aura fourni dans le fond la profusion qui fit défaut à la forme. Beaucoup de présentations, sur des sujets très variés, y compris en dehors de Perl, montrant que l'activité autour de Perl est loin de s'être tarie, et ce, à tous les niveaux. Enfin, de nombreux projets publics ou secrets sont en cours d'élaboration, entre autres pour relancer la conquête du monde par notre langage favori.

Avec la sortie de Perl 5.10 et probablement d'une bêta de Perl 6, l'année 2007 promet d'être riche et intéressante !

Sébastien Aperghis-Tramoni,

sebastien@aperghis.net – Sophia.pm

LIENS

- **Les Journées Perl 2006** :
  http://conferences.mongueurs.net/fpw2006/
- **Les supports des présentations** :
  http://conferences.mongueurs.net/fpw2006/talks
- **Les nombreuses photos** de la conférence publiées sur Flickr :
  http://flickr.com/photos/tags/fpw2006/interesting/
- QPSMTPd : **http://smtpd.develooper.com/**
- Djabberd :
  http://www.danga.com/djabberd/
- LORIA SynLex :
  http://www.loria.fr/~gardent/ladl/
- ALVIS : **http://www.alvis.info/**
- `DBIx::DataModel` :
  http://search.cpan.org/dist/DBIx-DataModel/
- Mason : **http://www.masonhq.com/**
- Catalyst : **http://www.catalystframework.org/**
- Parrot : **http://www.parrotcode.org/**
- Pugs : **http://www.pugscode.org/**

Jean-Claude Caty

EDIGÉO : échanger de l'information géographique

Les mondes de l'information géographique et du Logiciel libre font plutôt bon ménage ces derniers temps. Aux côtés des bases de données ouvertes à cette information particulière (comme le couple postgresql, postgis), fleurissent des SIG (Système d'Information Géographique comme grass ou qgis) et des serveurs d'application géographique (mapserver ou geoserver). L'administration fiscale, chargée de la gestion du plan cadastral français a d'ailleurs choisi des outils libres pour implémenter son futur serveur de plan.

Toutefois, le téléchargement des informations cadastrales fournira des fichiers au format d'échange EDIGÉO. EDIGÉO est une norme française pour l'échange de données géographiques (norme AFNOR NF52000 homologuée le 5 juillet 1999). Là, force est de constater que les outils libres permettant de récupérer ce format manquent. En effet, EDIGÉO est textuel, destiné à échanger de l'information et n'est pas d'une gestion pratique.

Votre serviteur a donc ouvert un projet relatif à cette norme afin de pallier ce manque. Un premier module, permettant de charger un échange EDIGÉO dans une base de données postgresql est très proche d'une version exploitable et permet déjà cette intégration.

Dans cette série d'articles, nous allons successivement survoler la norme EDIGÉO, puis balayer les outils réalisés et prévus dans le projet edigeo hébergé sur le serveur de développement collaboratif gna. Actuellement, 2 outils d'initialisation et de lecture existent, mais doivent être testés de manière approfondie (au moment où j'écris ces lignes).

La norme EDIGÉO : une structure en poupées russes

La norme EDIGÉO est un document de 318 pages. Nous ne ferons donc que survoler les concepts sans les détailler. Lorsque c'est possible, nous renverrons vers quelques références glanées sur internet.

Commençons par aborder EDIGÉO selon 2 angles de vues. Pratiquement tout d'abord en regardant un fichier; théoriquement ensuite en visitant les concepts.

1. Les fichiers EDIGÉO

Voici un exemple de fichier EDIGÉO :

```
BOMT 12:EDIGZASE.GEN - entête de métafichier; ligne #1
CSET 03:IRV - entête de métafichier; ligne #2
RTYSA03:DEG - descripteur d'étendue géographique
RIDSA14:EMPRISE_EDIGZA - identifiant du descripteur d'étendue
géographique
CM1CC22:+543674.75;+227667.90; - emprise : coordonnées minimales
CM2CC22:+545774.75;+229167.90; - emprise : coordonnées maximales
RTYSA03:GSE - descripteur de sous ensemble
RIDSA07:SeTOP_1 - identifiant du descripteur de sous ensemble
INFST00: - information sur le sous ensemble
STRSN01:1 - structure du sous ensemble; 1:topologique
REGSA00: - identifiant d'un éventuel calage
RTYSA03:GSE - descripteur de sous ensemble
RIDSA07:SeTOP_2 - identifiant du descripteur de sous ensemble
INFST00: - information sur le sous ensemble
STRSN01:1 - structure du sous ensemble; 1:topologique
REGSA00: - identifiant d'un éventuel calage
RTYSA03:GSE - descripteur de sous ensemble
RIDSA07:SeTOP_3 - identifiant du descripteur de sous ensemble
INFST00: - information sur le sous ensemble
STRSN01:1 - structure du sous ensemble; 1:topologique
REGSA00: - identifiant d'un éventuel calage
RTYSA03:GSE - descripteur de sous ensemble
RIDSA07:SeSPA_1 - identifiant du descripteur de sous ensemble
INFST00: - information sur le sous ensemble
STRSN01:3 - structure du sous ensemble; 3:spaghetti
REGSA00: - identifiant d'un éventuel calage
EOMT 00: -- fin de métafichier
```

A première vue, un échange EDIGÉO est composé d'un nombre variable de fichiers nommés `métafichiers` par la norme.

Chaque métafichier contient des `descripteurs`. Nous pouvons comparer un descripteur à une ligne d'une table dans une base de données. De la même manière qu'il existe plusieurs tables dans une base de données, les différents métafichiers définiront différents descripteurs. Notre exemple contient un descripteur d'étendue géographique (emprise) et 4 descripteurs de sous-ensembles géographiques.

Chaque descripteur est lui-même subdivisé en `champs`. Nous pouvons prolonger l'analogie précédente en comparant le champ d'un descripteur à une colonne dans notre base de données. La fin d'un descripteur est implicitement déterminée par le début d'un autre descripteur (champ `RTY`) ou une fin de métafichier (champ `EOM`).

L'ensemble des descripteurs sont encadrés par un en-tête et une fin de métafichier. L'en-tête est composé de 2 champs. Un champ `BOM` (*Begin Of Metafile*) et un champ `CSE` (*Character SEt*). La fin est composée du champ `EOM` (*End Of Metafile*).

Attardons-nous sur un champ `edigeo` :

```
RTYSA03:GTL
```

Ce champ a pour nom « `RTY` ». Un champ `RTY` est toujours le premier champ d'un descripteur. Sa valeur (ici `GTL`) précise le type du descripteur. Le type `GTL` caractérise un descripteur de lot (nous préciserons bientôt ce qu'est un lot dans un échange EDIGÉO).

Un champ est toujours composé de 6 parties. Voici les 6 parties de notre exemple :

PARTIE	SIGNIFICATION	TAILLE	VALEURS POSSIBLES
RTY	Nom du champ	3 caractères	
S	Nature du champ	1 caractère	C-Composé S-Simple, T-Réservé
A	Format du champ	1 caractère	<Espace>-Réservé A-Alpha, C-Coordonnée, D-Date, E-Réel avec exposant I-Entier signé N-Entier positif P-Référence vers un descripteur R-Réel sans exposant T-Texte
03	Longueur du champ	2 caractères	00 à 72
:	Séparateur	1 caractère	toujours « : »
GTL	Valeur du champ	Longueur	

La valeur d'un champ de format T peut intégrer des sauts de ligne (**0x0a**). C'est intéressant pour préciser une adresse ou toute autre information de plusieurs lignes.

La limite de 72 caractères peut être brisée en ajoutant un ou plusieurs champ `NEX` immédiatement derrière le champ à prolonger.

Le passage d'un champ à un autre sera déterminé par la première majuscule qui suit. Tout autre caractère est ignoré et peut être considéré comme commentaire. Tous les champs de notre exemple précédent de métafichier sont commentés.

Le projet edigeo hébergé par gna **[Lien 1] [Lien 2]** contient la description de la structure des métafichiers, des descripteurs et leurs champs dans différents scripts SQL **[Lien 3]**.

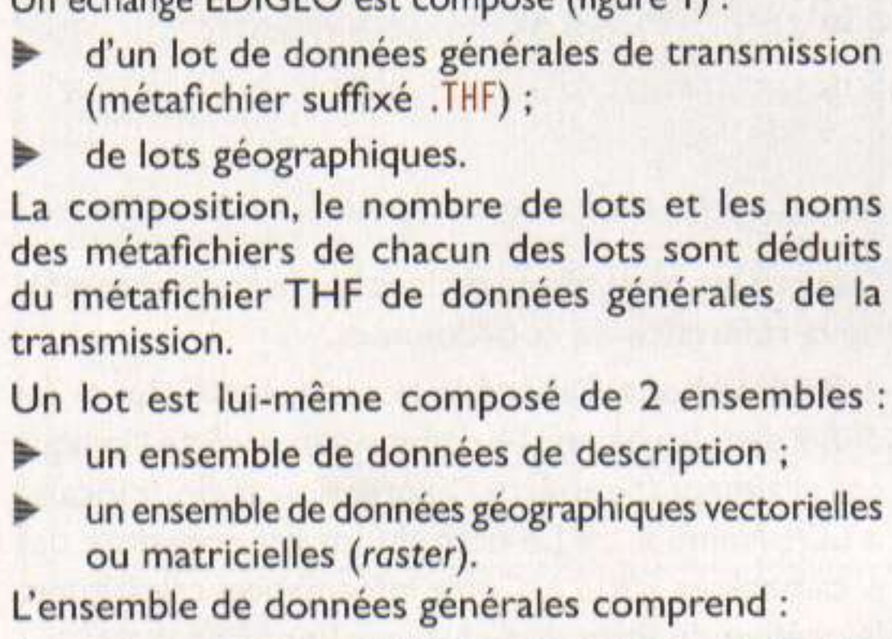
PARTIE	SIGNIFICATION
`metafichier.sql`	Description des métafichiers
`descripteur.sql`	Description des descripteurs
`champ.sql`	Description des champs

2. La structure EDIGÉO

L'organisation des informations

Commençons par regarder la structure, nous verrons ensuite le contenu des différents sous-ensembles constitutifs.

Figure 1 : Structure d'un échange EDIGÉO



Un échange EDIGÉO est composé (figure 1) :

- d'un lot de données générales de transmission (métafichier suffixé `.THF`) ;
- de lots géographiques.

La composition, le nombre de lots et les noms des métafichiers de chacun des lots sont déduits du métafichier THF de données générales de la transmission.

Un lot est lui-même composé de 2 ensembles :

- un ensemble de données de description ;
- un ensemble de données géographiques vectorielles ou matricielles (*raster*).

L'ensemble de données générales comprend :

- un sous-ensemble de données générales (métafichier suffixé `.GEN`) ;
- un sous-ensemble de la référence de coordonnées (métafichier suffixé `.GEO`) ;
- un sous-ensemble de description de la qualité des données (métafichier suffixé `.QAL`) ;
- un sous-ensemble de définition de la nomenclature (métafichier suffixé `.DIC`) ;
- un sous-ensemble de définition du schéma conceptuel des données (métafichier suffixé `.SCD`).

L'ensemble des données géographiques contient les objets géographiques eux-mêmes. Ces objets peuvent être :

- dans un sous-ensemble de données géographiques vectorielles (métafichier suffixé `.VEC`) ;
- dans un sous-ensemble de données géographiques matricielles (métafichier suffixé `.MAT`).

Les données cadastrales sont par exemples constituées de 4 sous-ensembles de données géographiques vectorielles :

- les sections cadastrales, données vectorielles topologiques (sans trous, ni recouvrements) ;
- les feuilles cadastrales, données vectorielles topologiques ;
- les parcelles, données vectorielles topologiques ;
- les autres objets cadastraux, données vectorielles spaghetti (sans contrainte de trou ou chevauchement).

3. Les sous-ensembles

Le sous-ensemble des données générales de transmission (.THF)

Les données générales de transmission définissent la structure générale de l'échange.

Celle-ci comprend donc un descripteur de support précisant notamment auteur et destinataire de l'échange et surtout le nombre de lots de cet échange.

Le support est suivi de n descripteurs de lots desquels sont déduits les noms des métafichiers de l'ensemble de données générales et la composition des sous-ensembles de données géographiques.

Pour connaître le nom du fichier d'un sous-ensemble, le principe est le suivant :

1. Repérer le lot qui nous intéresse dans le métafichier THF. Voici par exemple le début d'un descripteur de lot :

```
RTYSA03:GTL
RIDSA06:EDIGZA
LONSA06:EDIGZA
INFST00:
GNNSA02:SE
GNISA04:SeGN
GONSA02:SE
GOISA04:SeGO
```

Dans ce lot, nous allons rechercher le sous-ensemble de la référence de coordonnées.

2. Recherchons d'abord le nom du lot. Celui-ci est défini dans le champ LON comme son nom ne l'indique pas vraiment (peut-être l'expression anglo-française « LOt *Name* » ...). Le nom du lot est constitué des 6 caractères EDIGZA. Cette information constituera le préfixe du nom du fichier recherché.

3. Recherchons ensuite le nom du sous-ensemble de la référence de coordonnées. Vous trouverez cette information dans le champ GON, soit SE.

4. Enfin, nous savons tous que l'extension du fichier recherché est GEO. En concaténant ces différentes parties, nous déduisons que la référence de coordonnées se trouve dans le fichier EDIGZASE.GEO.

Le sous-ensemble des données générales (.GEN)

Le sous-ensemble de données générales précise l'emprise géographique des données du lot et détaille ensuite les sous-ensembles de données géographiques en y ajoutant leur structure (spaghetti, réseau, topologique ou matricielle). Nous n'entrerons pas plus dans le détail de ces structures. L'exemple de métafichier EDIGÉO vu précédemment était un sous-ensemble de données générales, constitué d'un descripteur d'étendue géographique et 4 descripteurs de sous-ensemble.

Le sous-ensemble de la référence de coordonnées (.GEO)

Le sous-ensemble de la référence de coordonnées indique dans quel système cartographique sont exprimées les coordonnées des données géographiques.

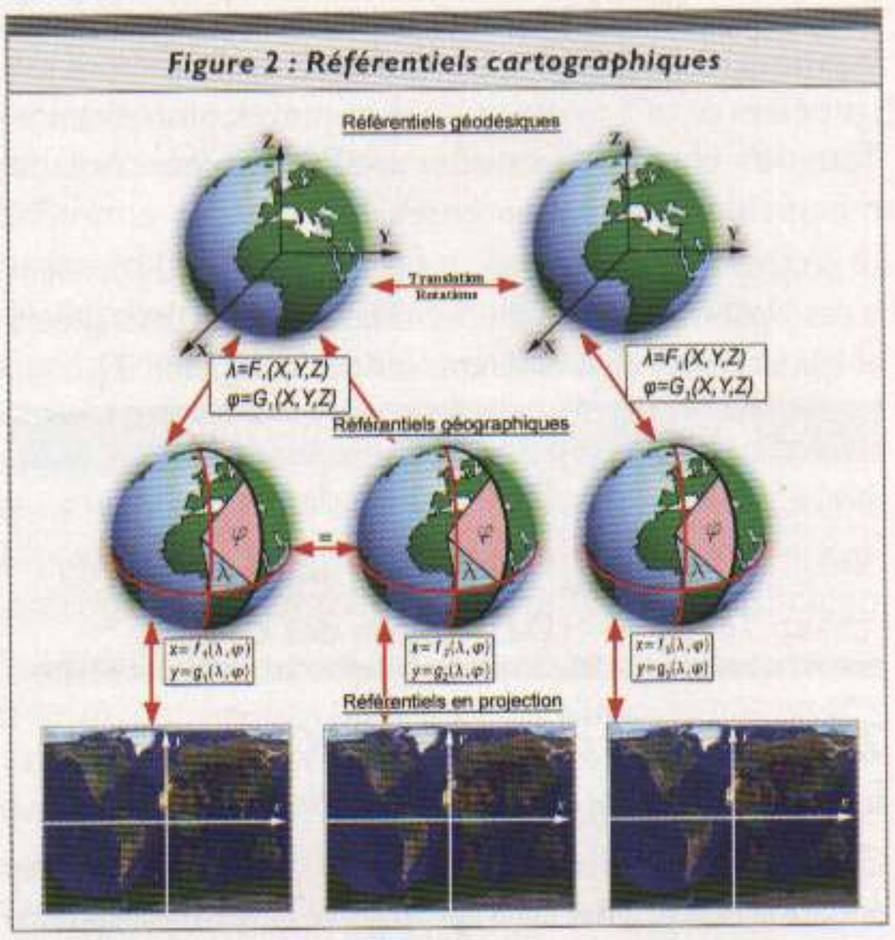
Les 4 principaux types de référentiels cartographiques sont les suivants :

- Les **référentiels géodésiques** (figure 2) avec origine au centre du globe terrestre. La précision des techniques de positionnement du centre de la terre s'améliorant au fil des années, il existe plusieurs référentiels avec différentes origines et orientations. De plus, les besoins cartographiques locaux conduisent à définir des référentiels mieux adaptés à ces besoins.
- Les **référentiels géographiques** (figure 2) – deuxième type de référentiel – s'appuient sur le précédent. Notre planète est un « patatoïde » (**géoïde** dans le jargon cartographique). Une modélisation mathématique de notre globe vers un ellipsoïde, volume proche du géoïde constitue le support du référentiel géographique. Les coordonnées sont alors constituées de 2 angles : latitude et longitude. Il est nécessaire de définir un méridien particulier pour l'origine des longitudes (Greenwich, Paris...).
- Les **référentiels en projection** (figure 2) s'appuient sur un ellipsoïde donné. Dans les usages courants, il n'est pas commode de devoir représenter une carte sur un ellipsoïde. Nous utilisons tous des cartes papier, représentations planes d'un ellipsoïde. Ces référentiels consistent donc à définir une transformation mathématique permettant d'exprimer les coordonnées planes en fonction des longitude et latitude.
- Les 2 derniers référentiels ne permettent pas de définir d'altitude. On peut donc leur adjoindre un système altimétrique.

Le CNIG (Conseil National de l'Information Géographique), organisme public, référence les systèmes cartographiques utilisables avec la norme EDIGÉO. Cette liste est disponible sur le serveur du CNIG. **[Lien 4]**

Ce même sous-ensemble définit d'éventuel descripteur de calage. Les coordonnées peuvent alors être exprimées dans un système de coordonnées local. Le descripteur de calage fournit les informations permettant la transformation du système local au système cartographique. Ceci est particulièrement utile aux informations matricielles.

Figure 2 : Référentiels cartographiques



Le sous-ensemble de description de la qualité (.QAL)

Le sous-ensemble de description de la qualité des données permet d'apporter des précisions qualitatives sur les données. Ce sous-ensemble en référence 9 types différents. Citons par exemple des informations relatives à la précision des données. D'autres relatives à la généalogie ou à l'actualité. Le lecteur souhaitant approfondir se reportera :

- au Bulletin d'Information de L'IGN n° 67 de Benoît David & Pascal Fasquel : « Qualité d'une base de données géographique : concepts et terminologie »
- à la thèse d'Atef Bel-Hadl-Ali relative à la qualité des données **[Lien 5]**.

Le sous-ensemble de définition de la nomenclature (.DIC)

Le sous-ensemble de définition de la nomenclature est un dictionnaire de données appelé « **nomenclature d'échange** ». La norme prévoit 3 types de nomenclatures :

- Une nomenclature **générale** commune à tous et gérée par le CNIG.
- Une nomenclature **sectorielle**, résultat de l'accord entre les collectivités ou personnes qui s'échangent des données géographiques. Cette nomenclature doit respecter la codification de la norme.
- Une combinaison des 2 types de nomenclatures.

La nomenclature générale est gérée par le CNIG et est évolutive. La version actuelle est disponible sur le serveur du CNIG **[Lien 6]**.

Le projet EDIGÉO inclut les référentiels et la nomenclature dans des tables SQL.

Le code d'un objet dans la nomenclature possède 4 composantes séparées par le caractère « _ ». Ces 4 composantes sont :

- Le **domaine** (1 lettre majuscule). Le code du domaine lui-même a ses autres composantes à « 0 ». Par exemple, le code du domaine « *SURFACE D'ACTIVITE ET BATI* » est « E_0_0_0 ».
- La **classe** (1 ou 2 chiffres). Le code de la classe elle-même a ses composantes suivantes à « 0 ». Par exemple, le code de la classe « *ZONE D'ACTIVITES SPORTIVES* » est « E_4_0_0 ».
- L'**objet générique** (1 à 3 chiffres). Le code de l'objet générique lui-même a sa dernière composante à « 0 ». Par exemple, l'objet générique « *STADE* » est « E_4_1_0 ».
- L'**objet** (1 à 4 chiffres). Par exemple, le code de l'objet « *PISTE D'ATHLETISME* » est « E_4_1_1 ».

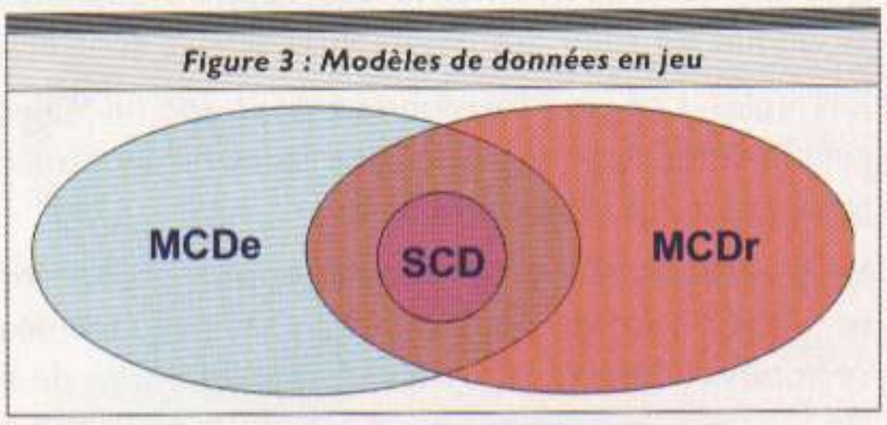
Le sous-ensemble de définition du schéma conceptuel des données (.SCD)

Le sous-ensemble de définition du schéma conceptuel des données est le modèle des données de l'échange.

C'est, en quelque sorte, l'équivalent d'un fichier XML schéma.

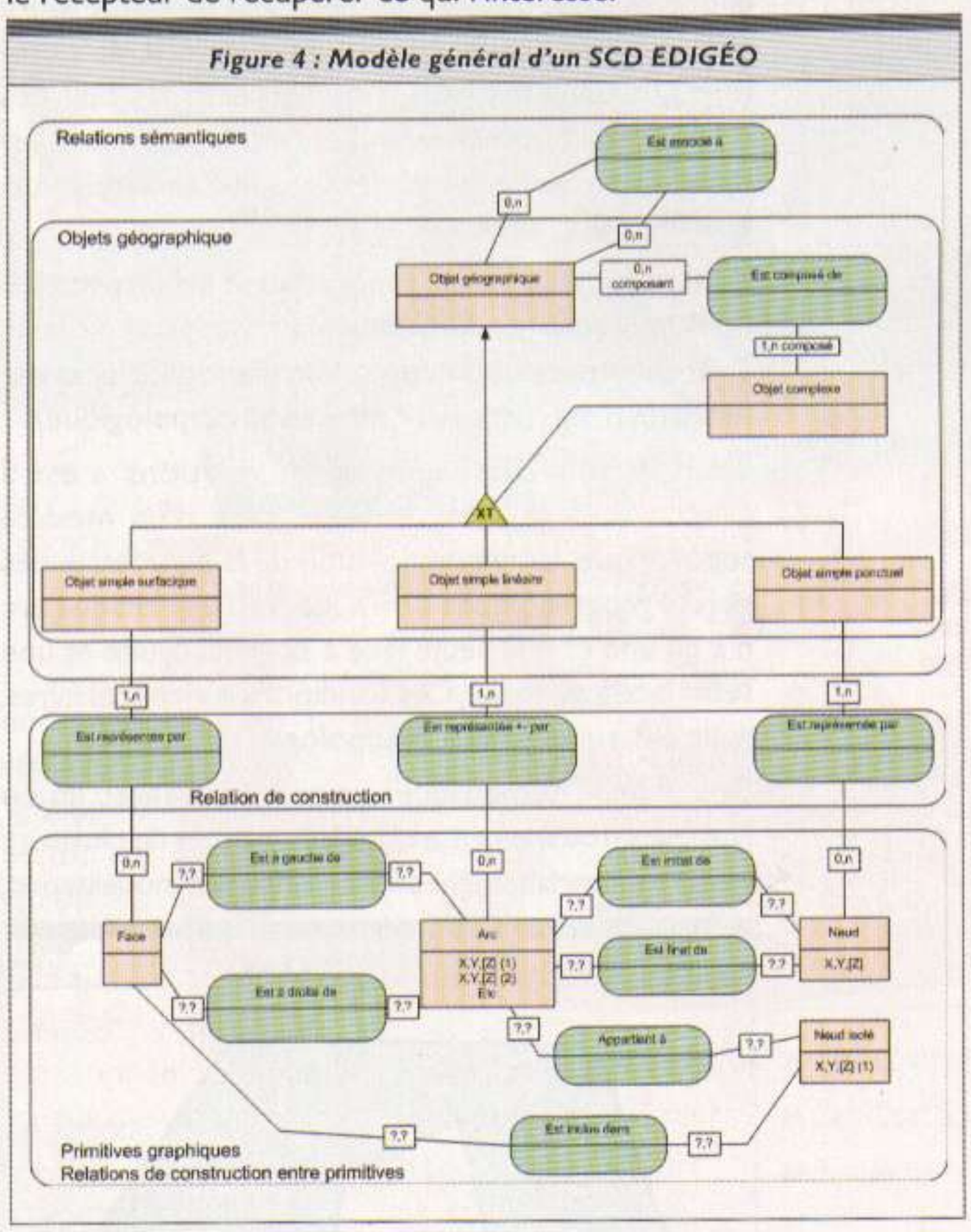
Lors du processus d'échange de données, l'émetteur devra extraire et convertir tout ou partie des données intéressant le destinataire. Émetteur et destinataire disposent de leur propre modèle de données. Si nous notons MCDe et MCDr les modèles de données respectifs de l'émetteur et du récepteur, le SCD EDIGÉO sera probablement un sous-ensemble de l'intersection de ces 2 modèles.

Figure 3 : Modèles de données en jeu



Le plus souvent, SCD résultera d'une convention entre l'émetteur et le récepteur. S'agissant de données gérées par un service public comme le cadastre, il s'agit plutôt de fournir les données publiques. A charge ensuite pour le récepteur de récupérer ce qui l'intéresse.

Figure 4 : Modèle général d'un SCD EDIGÉO



Par ailleurs, le modèle de données transmis devra obligatoirement se conformer au modèle général défini (figure 4) par la norme. Ce modèle dissocie géométrie et objets géographiques.

Les sous-ensembles des données vectorielles (.VEC)

L'ensemble des objets géographiques est partitionné en objets géographiques ponctuels, linéaires, surfaciques ou complexes. Sur le schéma, ce partitionnement est symbolisé par une relation d'héritage avec totalité et exclusion (XT selon la notation Merise).

La géométrie des objets simples est déterminée par les relations de constructions les liant aux primitives géométriques. Les objets ponctuels, linéaires et surfaciques sont respectivement liés à des nœuds, arcs et faces. La géométrie des objets complexes se déduit de celle des objets auxquels ils sont liés.

Exemple : L'objet géographique `pays` est un objet simple surfacique lié à plusieurs faces (le territoire principal et les différentes îles).

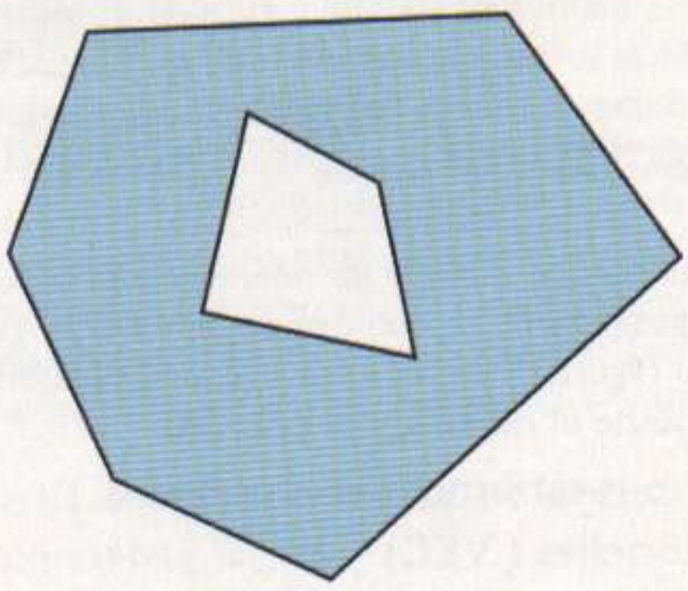
La géométrie est entièrement supportée par les primitives `nœud` et `arc`. Les faces sont déduites des relations « est à gauche de » et « est à droite de » les liant aux arcs des contours. Un arc peut être une ligne brisée, un arc de cercle ou une courbe. L'échange de courbes doit s'accompagner de conventions entre émetteur et récepteur. Le modèle de données de l'échange devra alors définir des attributs de l'arc précisant le type et les paramètres de la courbe, voire son équation si l'on dispose d'un module de calcul formel. L'usage est plutôt de convertir les courbes en une ligne brisée proche. Les calculs géométriques en sont grandement facilités sans que cela nuise à la qualité topographique.

Le schéma du modèle général laisse en suspend les cardinalités des relations entre primitives géométriques. Ces cardinalités sont en effet différentes selon le modèle utilisé (spaghetti, réseau ou topologique).

Considérons par exemple, les relations « est à gauche de » et « est à droite de » d'un modèle topologique. La décomposition de la géométrie des objets géographiques devra assurer que chaque arc n'a qu'une et une seule face à sa gauche, une et une seule face à sa droite. Ces conditions sont nécessaires, mais pas suffisantes à la topologie.

Des objets géométriques en couronne (avec un ou plusieurs trous) ou en archipel (constitués de plusieurs faces) sont parfaitement décrits dans cette modélisation.

Figure 5 : Objet à trou



Considérons par exemple une parcelle avec un trou (figure 5). La parcelle – objet simple surfacique – sera donc liée à une face. Cette face sera elle-même liée avec les arcs formant les contours intérieur et extérieur.

Conclusion

La norme EDIGÉO, en intégrant notamment le modèle des données de l'échange, permet d'automatiser de nombreux traitements, d'opérer un certain nombre de vérifications. Ses modèles géométriques, dissociant les objets de leur géométrie, permettent d'éviter des redondances.

Nous verrons donc, dans un prochain article, différents objectifs du projet edigeo. EDIGÉO étant destiné à l'échange de données, cela implique différents traitements pour intégrer un échange dans notre SIG. Ces traitements consistent principalement à placer les données dans une base transitoire, réaliser différents contrôles et exporter ensuite vers le SIG cible.

Les contrôles pourront permettre à tout producteur de données EDIGÉO de vérifier ses échanges avant transmission.

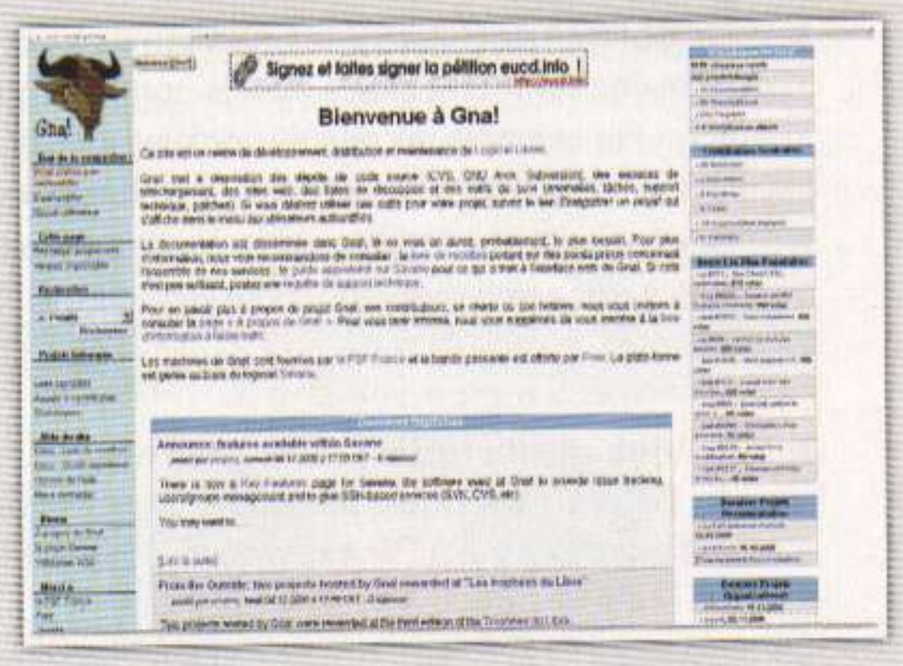
En recherchant EDIGÉO dans Google, on retrouve plusieurs contributions dans des forums montrant le manque d'outils, la méconnaissance de la norme. L'un de mes souhaits serait de fédérer les énergies autour du projet pour démocratiser cette norme malheureusement trop peu connue.

Jean-Claude Caty,

jc.caty@laposte.net

LIENS

- 1. Site de développement collaboratif gna : **https://gna.org**



- **2**. Projet EDIGÉO : **https://gna.org/projects/edigeo**
- **3**. Scripts SQL du CVS : **http://cvs.gna.org/cvsweb/edigeo/metaedigeo/sql/?cvsroot=edigeo**
- **4**. Systèmes cartographiques du CNIG : **http://www.cnig.gouv.fr/upload/ressource/r1091104706.PDF**
- **5**. Qualité des données géographiques : **http://www.univ-mlv.fr/recherche/WebTheses/UMLV-2004-000157.pdf**
- **6**. Nomenclature générale du CNIG : **http://www.cnig.gouv.fr/upload/ressource/r1091105628.ZIP**

Laurent Vivier

Principes des systèmes d'amorçage

A chaque fois que vous démarrez un système d'exploitation, vous l'utilisez ; peut-être même sans le savoir. Son nom est peut-être GRUB [1], LILO [2], ELILO [3], yaboot [4] ou encore BootX [5]. Nous allons voir ici quels en sont les principes en prenant pour exemple un des plus simples : EMILE [6]. Vous ne l'utiliserez sûrement jamais : l'architecture sur laquelle il s'utilise n'est plus commercialisée depuis 1995. En effet, EMILE est un chargeur d'amorçage pour linux/m68k [7] sur Macintosh à base de processeurs de la série 680x0.

Principes fondamentaux

La première des qualités d'un système d'amorçage (ou *bootloader*) est d'avoir la capacité de prendre le contrôle de l'ordinateur lorsqu'on l'allume. Ce résultat est généralement obtenu en plaçant le système d'amorçage dans le secteur d'amorçage du disque de démarrage. On verra que cela n'est pas aussi simple qu'on peut le penser. Ensuite, le rôle du système d'amorçage est de charger en mémoire l'image d'un noyau et quelquefois un RAMDISK (ou comme GRUB les nomme : des modules). Les plus complexes permettent à l'utilisateur de choisir parmi une liste préétablie. Une fois cette tâche effectuée, il doit placer le système dans un état permettant de démarrer le noyau tout en lui passant quelques paramètres, tels que la ligne de commande ou l'adresse du RAMDISK. À ce point, il ne lui reste plus qu'à passer la main : placer le pointeur d'instruction du processeur sur le point d'entrée du noyau chargé.

Prise de contrôle

La prise de contrôle se fait par l'intermédiaire du secteur d'amorçage. Le nombre et la taille des secteurs d'amorçage dépendent du média (disquette, disque dur, CD) et de l'architecture (PC, Mac ou autres). Nous allons étudier ici le cas du secteur d'amorçage d'une disquette pour Macintosh. La taille d'un secteur physique est 512 octets, et le secteur d'amorçage est composé de deux secteurs physiques. Nous avons donc 1024 octets pour stocker notre système d'amorçage... mais peut-on stocker un système d'amorçage digne de ce nom dans 1024 octets ? La réponse est malheureusement « non ». Alors, comment faire ?

Premier niveau

La solution consiste à diviser le chargeur d'amorçage en deux parties : un « niveau 1 » et un « niveau 2 ». Le rôle du niveau 1 est de charger le niveau 2 tout en étant assez petit pour être contenu dans le secteur d'amorçage. Pour illustrer cela, nous allons prendre pour exemple le niveau 1 d'EMILE.

```
101 start:
102     /* Allocate Memory for second stage loader */
103
104     lea         ioReqCount(%pc),%a0
105     move.l      (%a0), %d0
106     add.l       #4, %d0
107     NewPtr
108     move.l      %a0, %d0
109     add.l       #3, %d0
110     and.l       #0xFFFFFFFC.l, %d0
111
112     /* save result in the ParamBlockRec.ioBuffer */
113
114     lea         ioBuffer(%pc),%a0
115     move.l      %d0,(%a0)
116
117     /* Now, we load the second stage loader */
118
119     lea         param_block(%pc),%a0
120     PBReadSync
121
122     /* call second stage bootloader */
123
124     move.l      ioBuffer(%pc),%a0
125     jmp         (%a0)
```

Comme on le voit, le principe est simple : on alloue la mémoire (`NewPtr`) et on charge en mémoire (`PBReadSync`) le niveau 2. Ces opérations sont effectuées en faisant appel à deux *traps* contenus dans la ROM du Macintosh :

```
19 .macro NewPtr
20      .short 0xA11E
21 .endm
22
23 .macro PBReadSync
24      .short 0xA002
25 .endm
```

Les informations à connaître pour être capable de charger le niveau 2 sont sa taille et sa position sur la disquette. Ces informations sont, elles aussi, stockées dans le secteur de démarrage, directement dans la structure utilisée par `PBReadSync` pour lire les secteurs de données (`param_block`). `IoReqCount` est la taille, `ioBuffer` contient l'adresse en mémoire où stocker le niveau 2. Une fois les données chargées, le niveau 1 passe la main au niveau suivant en sautant au point d'entrée du niveau 2.

Second niveau

Le niveau 2 est donc le système d'amorçage en lui-même. C'est avec lui que vous allez interagir, pour modifier la ligne de commande par exemple. C'est lui qui va préparer la machine et charger les fichiers nécessaires en mémoire. Il s'exécute en mode « superviseur » et a donc tous les droits.

Lecture des fichiers

Maintenant que le système d'amorçage a pris le contrôle de la machine, il doit mettre en place en mémoire les différentes structures. Pour cela, il faut charger à partir du disque l'image du noyau. La lecture de cette image à partir de la disquette se faisant de la même manière que l'a fait le niveau 1 pour lire le niveau 2, nous allons plutôt nous attarder sur la lecture d'un fichier à partir d'un disque dur.

La problématique ici est de retrouver les données d'un fichier. En effet, celles-ci ont été stockées à divers endroits du disque par le système de fichier en fonction des blocs de données disponibles dans la partition. Il existe deux solutions à ce problème : la première est d'ajouter un pilote permettant de lire le fichier en décodant la structure du système de fichiers, la seconde consiste à encoder la liste des blocs à lire dans le système d'amorçage. L'inconvénient de la première solution est qu'il existe différents types de système de fichiers et donc différents algorithmes pour accéder aux données du fichier. Des systèmes d'amorçage comme GRUB ont pris le parti d'intégrer cette méthode. EMILE utilise la seconde méthode, plus simple et déjà éprouvée par LILO, mais qui a l'inconvénient de nécessiter la mise à jour du niveau 2 à chaque fois que l'image du noyau est modifiée (lors d'une mise à jour du système, par exemple). Cette méthode nécessite la présence d'un outil tournant sous le système cible. Cet outil a pour tâche d'établir la liste des blocs utilisés par le fichier et de l'encoder dans le niveau 2. La difficulté ici est d'établir cette liste sans avoir à décoder directement le système de fichier ; si c'était le cas, nous nous retrouverions dans le premier cas de figure. Heureusement, Linux fournit un `ioctl()` permettant de récupérer le numéro de bloc physique dans la partition à partir du numéro de bloc logique dans le fichier. Le principe est le suivant :

```
142   /* get first physical block */
143
144   last_physical = 0;
145   ret = ioctl(fd, FIBMAP, &last_physical);
146   if (ret != 0) {
147           perror("ioctl(FIBMAP)");
148           return -1;
149   }
150
151   zone = last_physical;
152   aggregate = 1;
153
154   /* seek all physical blocks */
155
156   current = 0;
157   for (logical = 1;
158        logical < (st.st_size + block_size - 1) / block_size;
159        logical++) {
160           physical = logical;
161           ret = ioctl(fd, FIBMAP, &physical);
162           if (ret != 0)
163                   break;
164           if (physical == last_physical + 1) {
165                   aggregate++;
166           } else {
167                   ADD_BLOCK(first_block + zone * sectors_per_block,
168                             aggregate * sectors_per_block);
169                   zone = physical;
170                   aggregate = 1;
171           }
172           last_physical = physical;
173   }
174
175   ADD_BLOCK(first_block + zone * sectors_per_block,
176             aggregate * sectors_per_block);
```

On appelle donc `ioctl()` avec la requête `FIBMAP` qui prend en entrée le descripteur du fichier à analyser et un pointeur sur une variable contenant le numéro de bloc logique, en sortie ce pointeur pointera sur le numéro de bloc physique dans la partition. La particularité de l'algorithme précédent est qu'il essaye d'encoder des séquences de blocs consécutifs. De plus, comme le système d'amorçage n'a pas la connaissance des partitions, l'outil doit ajouter au numéro de bloc obtenu un incrément correspondant au début de la partition (`first_block`) pour avoir un numéro de bloc relatif au début du disque. Le début de partition est fourni par une autre requête `ioctl()` avec le paramètre `HDIO_GETGEO` :

```
70    ret = ioctl(fd, HDIO_GETGEO, &geom);
71    if (ret == -1)
72    {
73            fprintf(stderr, "%s: ioctl(HDIO_GETGEO) fails: %s",
74                            dev_name, strerror(errno));
75            return -1;
76    }
...
88    *first_block = geom.start;
```

La liste des blocs est ensuite écrite à un endroit défini dans le niveau 2 (et est donc chargée en même temps que le niveau 2 par le niveau 1). Lorsque le niveau 2 va vouloir trouver le bloc physique du disque associé à un numéro de bloc logique du noyau (ou du RAMDISK). Il va utiliser cette liste avec l'algorithme suivant :

```
15 static unsigned long seek_block(container_FILE *file)
16 {
17     struct emile_container *container = file->container;
18     ssize_t current;
19     int i;
20     unsigned long offset = file->offset;
21     int block_size = file->device->get_blocksize(file->device->data);
22
23     for (i = 0, current = 0;
```

```
24          container->blocks[i].offset != 0; i++)
25      {
26              int extent_size = block_size *
27                                  container->blocks[i].count;
28
29              if ( (current <= offset) && (offset < current + extent_size) )
30              {
31                      return container->blocks[i].offset +
32                                  (offset - current) / block_size;
33              }
34
35                  current += extent_size;
36      }
37
38      return 0;
39  }
```

`file->offset` est la position logique dans le fichier. En retour, la fonction est égale au numéro de bloc physique correspondant. La liste des blocs précédemment calculée est stockée dans le tableau `contrainer->blocs`.

Le système d'amorçage procédera de la même manière s'il doit charger un RAMDISK. La seule différence notable entre le noyau et le RAMDISK est que le noyau est décompressé par le système d'amorçage alors que le RAMDISK l'est par le noyau.

Passage de paramètres

Il y a au moins deux paramètres standards à passer au noyau : la ligne de commande, et, quelquefois, l'adresse du RAMDISK en mémoire. Alors que GRUB utilise une structure de données assez primitive au format *multiboot* **[8]**, les noyaux pour architecture « m68k » utilisent des *tags* plus évolués. Pour ceux qui ont connu l'Amiga (une machine de l'architecture m68k) dans les années quatre-vingt, la notion de « tags » leur rappellera étrangement les *chunks* du format IFF **[9]**. Le système d'amorçage passe au noyau un pointeur sur le *boot record* contenant une suite de tags. Chaque tag est identifié par un type. Il contient sa longueur qui permet de trouver le tag suivant et d'ignorer le courant si le noyau ne le connaît pas. Un boot record peut donc contenir la ligne de commande (tag de type `V2_BI_COMMAND_LINE`) et un autre l'adresse du RAMDISK (tag de type `V2_BI_RAMDISK`). Vous me direz que c'est bien compliqué pour passer deux informations... effectivement, mais j'ai passé sous silence une phase qui se déroule en début de niveau 2 : l'identification des ressources de la machine. Une des tâches du système d'amorçage est d'identifier le type de la machine (architecture, type de processeur, présence de coprocesseurs) et les ressources présentes (port série, adresse de la mémoire vidéo, organisation de la mémoire vive...). Cette tâche est aisée pour le système d'amorçage, puisqu'il peut encore consulter les routines de la ROM, ce que ne peut pas faire le noyau, une fois démarré. L'emploi des tags permet de passer toutes ces informations d'une manière générique au noyau. Une fois la structure d'information de démarrage initialisée, elle est placée immédiatement après le noyau, ce qui lui permettra de la retrouver.

Démarrage

Tout est en place, il ne reste plus qu'à passer la main au noyau en plaçant le pointeur d'instruction du processeur sur son point d'entrée. En fait, ce n'est pas aussi simple : « avant de sortir, éteignez la lumière ! », avant de passer la main au noyau, il faut bien veiller à mettre un terme à toutes les activités du système natif, et, plus particulièrement, à ses gestionnaires d'interruptions. Une autre des difficultés spécifiques à cette architecture est la présence et l'utilisation de la MMU du processeur : le système d'amorçage s'exécute avec la mémoire virtuelle activée. Il y a donc translation entre les adresses connues par le système d'amorçage et les adresses réelles de la mémoire physique. Et malheureusement, le noyau doit s'exécuter avec la mémoire virtuelle désactivée. Il faut donc introduire une phase de *bootstrap* qui va désactiver la mémoire virtuelle et recopier le noyau à l'adresse physique adéquate (que l'on connaît en décodant le format ELF du noyau), puis faire le saut dans le noyau.

Vous êtes maintenant dans le monde du noyau... et c'est une autre histoire !

Laurent Vivier,

laurent@lvivier.info

RÉFÉRENCES

- **[1]** http://www.gnu.org/software/grub/
- **[2]** http://lilo.go.dyndns.org/
- **[3]** http://elilo.sourceforge.net/
- **[4]** http://yaboot.ozlabs.org/
- **[5]** http://penguinppc.org/bootloaders/bootx/
- **[6]** http://emile.sourceforge.net/
- **[7]** http://www.debian.org/ports/m68k/
- **[8]** http://www.uruk.org/orig-grub/boot-proposal.html
- **[9]** http://en.wikipedia.org/wiki/Interchange_File_Format

Denis Bodor

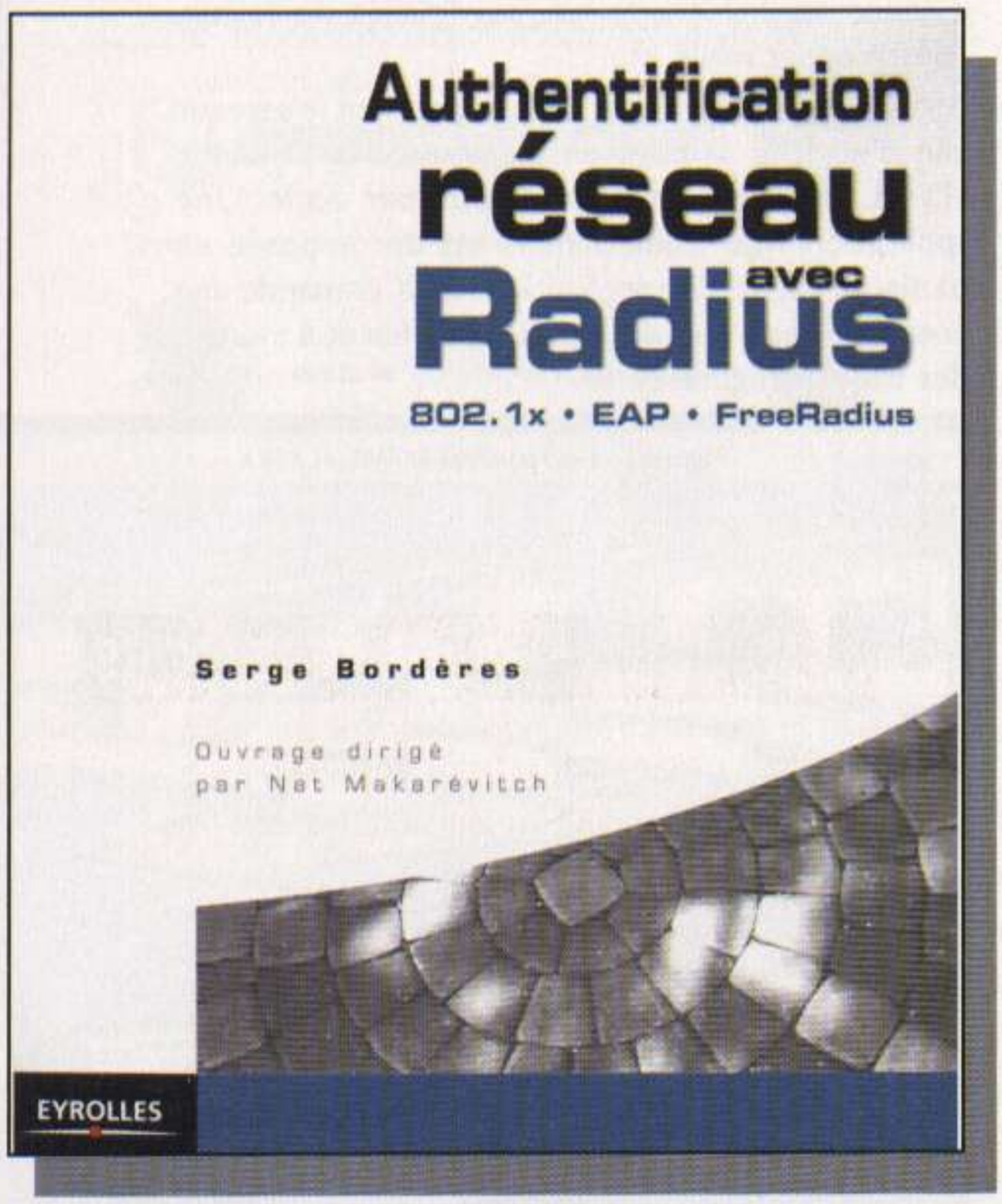
Authentification réseau avec Raduis

RADIUS ou *Remote Authentication Dial-In User Service* est un protocole client-serveur permettant la centralisation de l'authentification. Initialement utilisé par les fournisseurs d'accès à Internet pour identifier les utilisateurs des connexions par modem, Radius a rapidement trouvé son chemin vers l'administrateur système souhaitant simplifier les procédures d'authentification. Ceci est d'autant plus vrai depuis l'arrivé des connexions Wifi et de la mobilité IP en générale.

Le principal intérêt de RADIUS est de permettre à plusieurs serveurs l'accès à une base de données unique contenant les informations sur les utilisateurs. Le plus souvent, il s'agit d'une base LDAP.

Ce livre introduit le sujet par ses bases théoriques avant d'orienter rapidement les explications vers des standards dérivés comme 802.1X et EAP (*Extensible Authentication Protocol*). Côté implémentation, c'est, bien entendu, FreeRadius qui sera détaillé dans un contexte très Wifi. L'aspect pratique et tutoriel couvre aussi bien la configuration du serveur et des bornes d'accès que celle des clients ou la mise en place des certificats (pour l'authentification TLS). Une courte introduction aux PKI (IGC) permet d'utiliser une structure de configuration cohérente et facilement extensible.

Le livre, de manière générale, n'est pas orienté spécifiquement vers GNU/Linux, mais notre cher OS partage équitablement la vedette avec le principal système du marché.



Voici le sommaire :

- Pourquoi une authentification sur réseau local ?
- Matériel nécessaire
- Critères d'authentification
- Principes des protocoles Radius et 802.1X
- Description du protocole Radius
- Les extensions du protocole Radius
- FreeRadius
- Mise en œuvre de FreeRadius
- Configuration des clients 802.1X
- Mise en œuvre des bases de données externes
- Outils d'analyse
- Références
- La RFC 2865 – RADIUS

✓	Auteur : Serge Bordères
✓	ISBN : 2-212-12007-9
✓	Prix : 35,00 EUR

✓	Format : *210* pages
✓	Éditeur : Eyrolles

Philippe Prados

Développement web avancé avec AJAX

AJAX (Asynchronous JAvascript and Xml) est un terme inventé par Jesse James Garret [1] en février 2005 [2] pour désigner une technologie ancienne de 1999. Il s'agit d'utiliser du code Javascript sur le client pour invoquer le serveur en tâche de fond et obtenir des données au format XML afin de modifier le Dynamic HTML et rafraîchir une partie de la page.

Cela consiste à exploiter l'objet `XMLHttpRequest`, présent dans les navigateurs récents. Ce composant permet de jouer le rôle d'un client HTTP, d'envoyer une requête et d'attendre la réponse pour pouvoir la manipuler.

AJAX permet d'optimiser le trafic en ne ramenant que les données nécessaires au rafraîchissement d'une page et non une page entière. De vingt à quatre-vingt pourcents d'une page sert à la présentation. AJAX permet d'optimiser les affichages de type « liste/détail », de trier les données sur le client, d'aider l'utilisateur lors des saisies, d'enrichir les arbres ou d'alimenter les onglets... AJAX permet également d'offrir une approche « une seule page pour l'application ». Pour, par exemple, proposer un tableur ou un traitement de texte dans une page WEB.

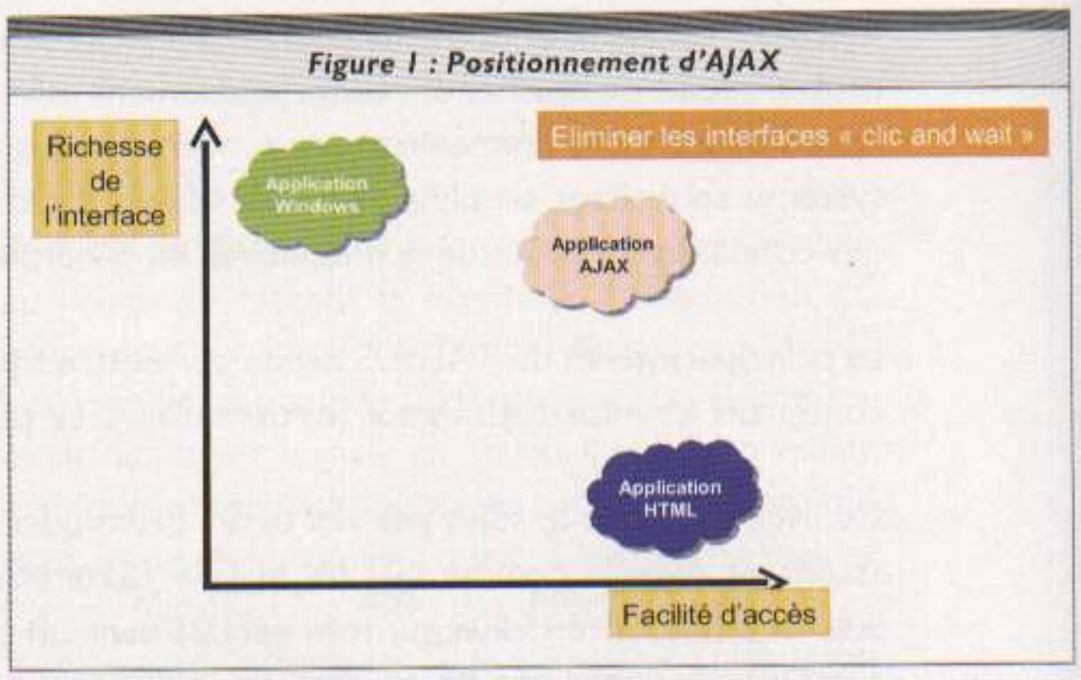
Sous le terme AJAX, on intègre également d'autres utilisations du Javascript. Il s'agit d'améliorer l'ergonomie de l'application en ajoutant des composants graphiques complémentaires (aide à la saisie, saisie de date, arbre, onglet, menus, piles, accordéon, curseur, barre de progression, etc.)

Si toutes les données sont disponibles, les technologies *Dynamic* HTML (DHTML) et CSS permettent une mise jour dynamique de la structure de la page. Il est possible de proposer une ergonomie proche de celle d'une application cliente classique. Des parties de pages sont cachées ou affichées à la demande ; des fenêtres apparaissent ou disparaissent sur évènements ; des zones sont complétées dynamiquement à l'insu de l'utilisateur, des arbres s'enrichissent au fur et à mesure, etc. Si les données ne sont pas disponibles, une requête AJAX asynchrone y remédiera.

Cette technologie a pour but d'améliorer l'efficacité des internautes et l'attractivité des sites. En réduisant de quelques secondes les manipulations des utilisateurs, un centre d'appel améliore sa rentabilité. En attirant de nouveaux internautes, un site améliore son ROI.

Les applications traditionnelles offrent une très grande richesse de l'interface. Des logiciels comme Open Office proposent des fonctionnalités que l'on ne retrouve pas sur les sites internet. À l'opposé, les applications HTML offrent une très grande facilité d'accès. AJAX est une technologie permettant de rapprocher ces deux objectifs (figure 1).

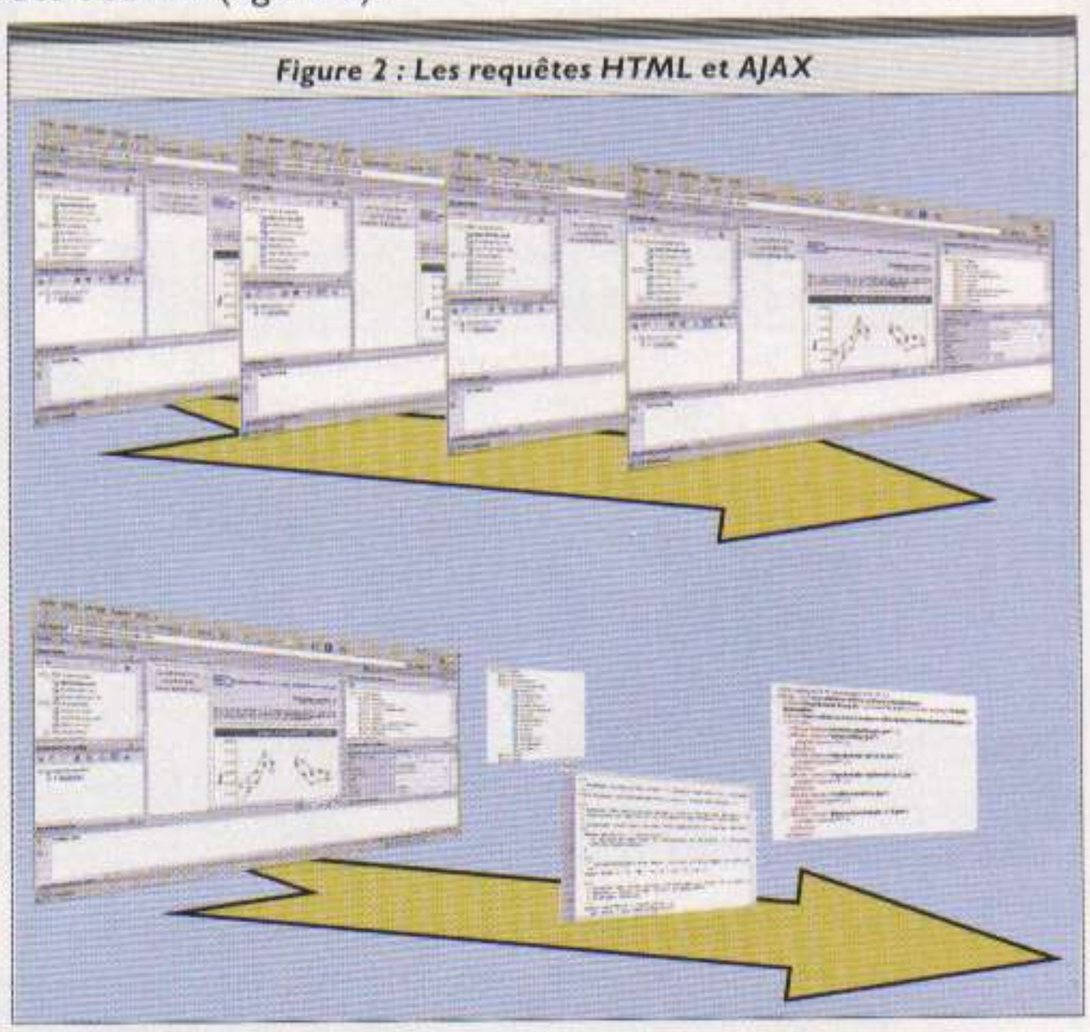
Figure 1 : Positionnement d'AJAX



De plus en plus de sites proposent des ergonomies enrichies. Citons : **www.netvibes.com**, **www.gmail.com**, **local.google.com**, **www.kiko.com**, **www.zimbra.com**, etc. L'objectif est de s'affranchir de l'ordinateur. Tous les fichiers et les applications seront disponibles sur une ou plusieurs machines du réseau. À partir de n'importe quel poste, muni d'un simple navigateur, il sera possible de retrouver son environnement, d'éditer ses fichiers textes, ses feuilles de calculs, consulter ses mails, etc.

AJAX permet d'invoquer dynamiquement le serveur, afin d'enrichir la page en manipulant le Dynamic HTML. La page est mise à jour par zone. Une application web traditionnelle est décomposée en plusieurs pages. Une application AJAX demande une première page, puis des données au fur et à mesure des besoins (figure 2).

Figure 2 : Les requêtes HTML et AJAX



Avant, le navigateur présente un contenu ; le serveur héberge une application et les requêtes au serveur sont explicites. Avec AJAX, le navigateur héberge

une application ; le serveur fournit des données et la plupart des requêtes au serveur sont implicites.

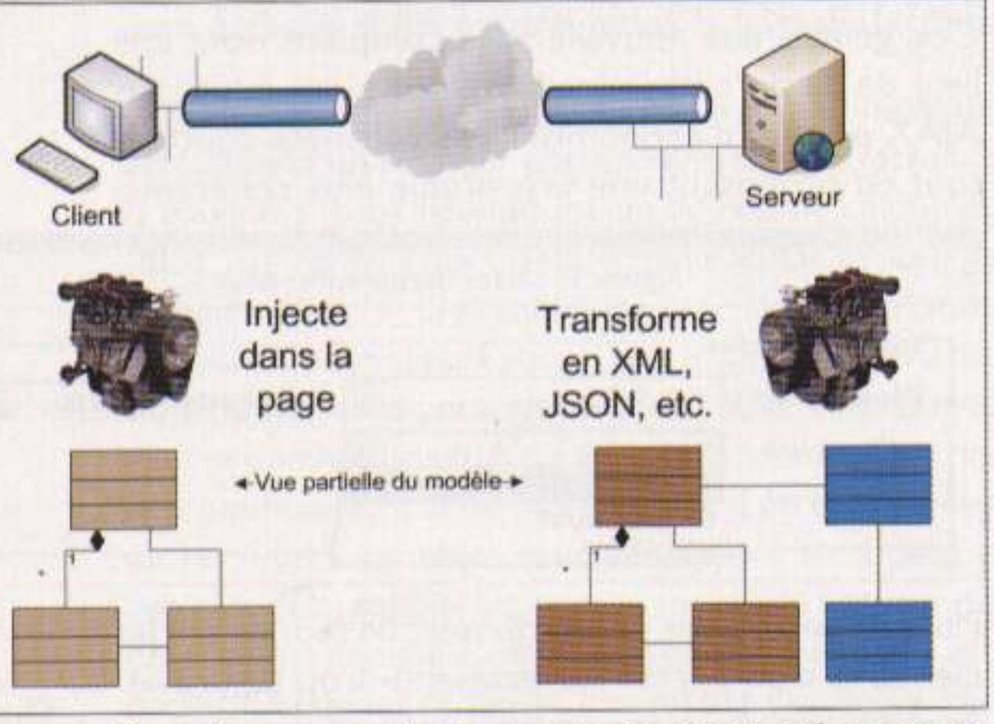
1. Comment cela fonctionne-t-il ?

Le rôle du serveur consiste à maintenir l'ensemble des objets de l'application et à envoyer quelques extraits au client. Dans la figure 3, le serveur transforme une partie des objets métier qu'il possède (les objets en marron) et les envoie au client sous un format compatible avec le HTTP. Le client transforme le flux en objets Javascript et manipule la page. Le client prend en charge quelques objets et s'occupe de les présenter. Il apporte alors des modifications à certains objets et les propage au serveur. Celui-ci peut alors décider ou non de sauver les modifications.

Nous revenons à une architecture de type client/serveur.

Il est préférable d'avoir dans le client un moteur générique de transformation du modèle vers du DHTML. Cela permet de généraliser le lien entre le modèle de donnée présent dans le client et la présentation. Beaucoup de *frameworks* AJAX travaillent sur cette approche. Ils utilisent des règles de transformations ou des annotations du XHTML. Certains proposent une connexion automatique entre les éléments de présentation et les objets présents sur le serveur.

Figure 3 : Export d'objets



Pour la communication entre le client et le serveur, il y a cinq étapes :

1. Création de paramètre de la requête pour le serveur.
2. Invocation de requêtes HTTP via l'objet `XMLHttpRequest`.
3. Attente de la réponse du serveur.
4. Analyse et transformation du résultat.
5. Modification dynamique de la page.

Reprenons les différentes étapes pour voir les stratégies possibles d'implémentations.

Pour la création de paramètre de la requête, il y a plusieurs approches :

- Création d'une URL de type GET. Les paramètres sont présents dans l'URL, séparés par un &.
- Récupération du formulaire HTTP. La requête HTTP est du type POST.
- Sérialisation d'objet Javascript. Il existe de nombreux formats (JSON, XML, CSV, etc.).
- Création d'une requête SOAP.

Ensuite, il faut invoquer le serveur. L'objet `XMLHttpRequest` est présent dans les navigateurs récents. Cela représente 92% des navigateurs en juillet 2006 d'après l'analyse effectuée par **www.thecounter.com** **[3]**. L'invocation est asynchrone. Il faut enregistrer un traitement qui sera invoqué lorsque la réponse est disponible.

Il reste à attendre que la requête soit terminée. Cela peut arriver immédiatement, mais parfois, lorsque le réseau n'est pas rapide ou que l'internaute utilise un modem, la réponse peut arriver plusieurs secondes après l'émission. Les réponses peuvent même ne jamais arriver ou arriver dans un ordre différent. Il faut alors traiter l'erreur.

Le serveur qui reçoit la requête est libre de retourner une réponse dans le format qu'il souhaite. Il peut retourner des extraits de pages HTML, des scripts JSON (*JavaScript Object Notation*, un code Javascript permettant lors de son exécution de construire une structure complexe d'objets), des flux XML (Content-Type : text/xml), des flux SOAP pour les services Web, du code Javascript ou un format propriétaire.

Comment le serveur peut-il générer ces différents formats ? Le tableau 1 propose différentes pistes.

Tableau 1 : Production des réponses AJAX

Format à produire	J2EE	.NET
TEXT	Servlet	ASP
HTML	JSP/JSF	ASP
XML	JAXB, Mapping DB/XML	.NET, Mapping DB/XML
JSON	Api Open source [4]	.NET
SOAP	EJB, Axis [5]	.NET
Javascript	JSP/JSF, Fwk	ASP, Fwk

Il faut ensuite transformer la réponse pour modifier la présentation. Si le serveur retourne des extraits de pages XHTML, il suffit d'inclure la réponse directement dans un tag `<div/>` via l'attribut `innerHTML`. Si le serveur retourne un flux JSON, il faut exécuter celui-ci avec la fonction `eval()` pour obtenir des grappes d'objets Javascript (attention à la sécurité), puis utiliser ces données pour manipuler le Dynamic HTML avec un script spécifique. Les méthodes `createElement()`, `createTextNode()` et `appendChild()` sont alors sollicitées.

Si le serveur retourne un flux XML, la première approche consiste à analyser les données pour manipuler le Dynamic HTML. La deuxième approche, plus simple, consiste à invoquer une transformation XSLT sur le client avec les données, pour générer un extrait XHTML et inclure le résultat dans un tag `<div/>` via l'attribut `innerHTML`.

L'approche Javascript permet d'animer la page, mais présente l'inconvénient d'être difficilement portable. Il

faut tester, re-tester le code avec différents navigateurs et différentes versions du même navigateur.

L'approche XSLT utilise le moteur XSLT présent dans les navigateurs. Elle permet de générer des portions de pages à partir de données au format XML. La transformation génère du XHTML à insérer dans la page. Cette approche permet de facilement trier ou filtrer les données en mémoire. Par exemple, le clic sur la tête d'une colonne permet de modifier le critère de tri du filtre XSLT. Les données sont, à nouveau, transformées en XHTML, mais avec un ordre différent. Le résultat remplace le tableau avec les nouvelles données. C'est le moteur XSLT du navigateur qui s'est chargé de pratiquement tout. Il est facile d'utiliser XPath pour naviguer dans des structures complexes et de sélectionner les données à manipuler. Comme le moteur XSLT est compilé, il est beaucoup plus rapide qu'un équivalent en Javascript.

Les scripts XSLT peuvent être réutilisés dans plusieurs pages. Ils peuvent faire partie de la charte graphique du site, au même titre que la feuille de style. Ainsi, tous les tableaux, par exemple, auront le même look et le même comportement.

Si le serveur retourne une réponse SOAP, il est préférable d'avoir préalablement généré un proxy **[6]** Javascript d'invocation du service Web. Ainsi, l'invocation d'une fonction Javascript permet de formater la question et d'analyser la réponse. Des outils **[7]** se proposent de générer cela à partir de la description du service WEB en WSDL. La difficulté avec cette approche est que la réponse arrive de façon asynchrone. Le code peut alors analyser les objets Javascript et manipuler le Dynamic HTML.

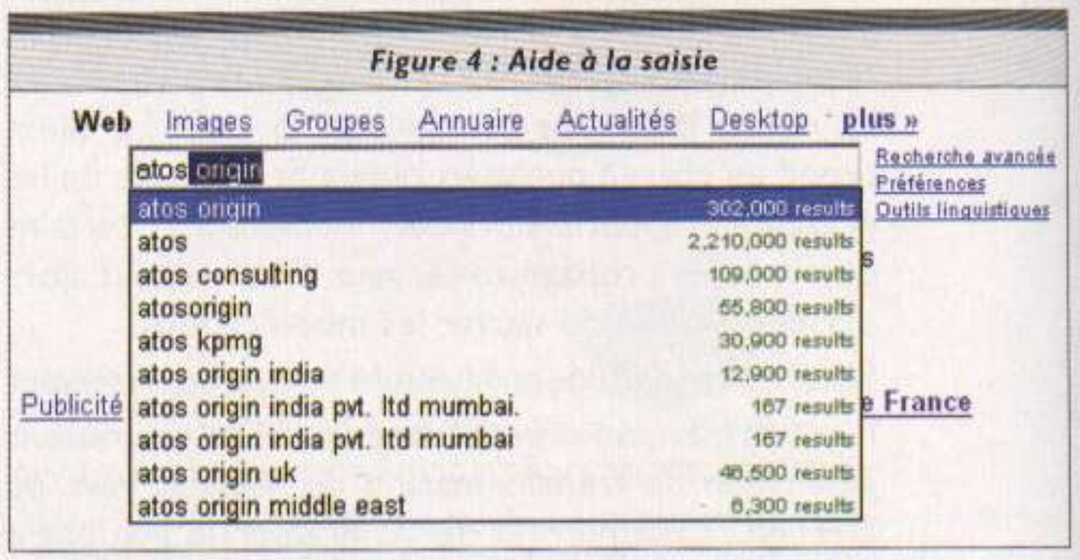
Si le serveur retourne directement un script Javascript, il suffit d'invoquer `eval()` avec le résultat. Il faut faire très attention au risque de sécurité que cela peut engendrer.

Certains frameworks permettent d'apporter des modifications à la présentation à partir de critères de sélection type XPath ou CSS **[8]**. Ainsi, une règle peut demander que tous les tags de type `<p/>` doivent avoir leurs styles passés de « caché » à « visible » et inversement.

Marier l'invocation asynchrone du serveur et la génération dynamique de la page permet d'offrir une navigation visuelle (*tooltips*, fenêtre flottante), une saisie interactive (auto-complétion, vérification dynamique, *drag-and-drop*) ou des animations (effet visuel). Les données non disponibles pour manipuler le Dynamic HTML sont demandées au serveur lorsque cela est nécessaire (approche paresseuse). Le modèle objet dans la mémoire du navigateur peut alors s'enrichir dynamiquement par l'ajout de branches à un arbre par exemple.

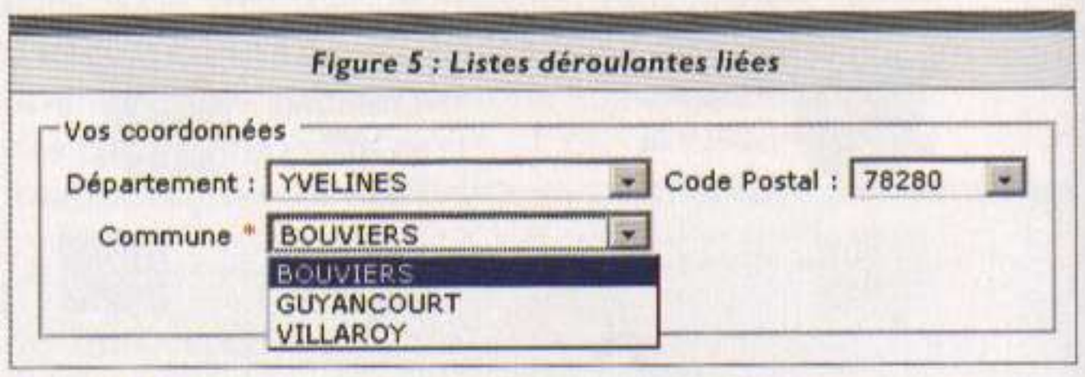
On rencontre de plus en plus d'aide à la saisie lorsque le nombre de valeur possible est important, mais limité. Par exemple, la liste des communes françaises est connue. Après quelques caractères, une requête part vers le serveur pour lui demander de sélectionner les candidats compatibles avec la saisie. Une liste déroulante est alors alimentée avec les dix candidats les plus courants et une suite à la saisie est proposées. La première valeur est ajoutée à la saisie. L'ajout est pré-sélectionné afin que la saisie d'un nouveau caractère efface la proposition (figure 4). L'appui d'un nouveau caractère améliore la sélection en interrogeant à nouveau le serveur ou en filtrant la sélection précédente. Lorsqu'il ne reste qu'un choix, un bip retentit au premier caractère invalide. Cela est également utilisé pour les moteurs de recherche.

Figure 4 : Aide à la saisie



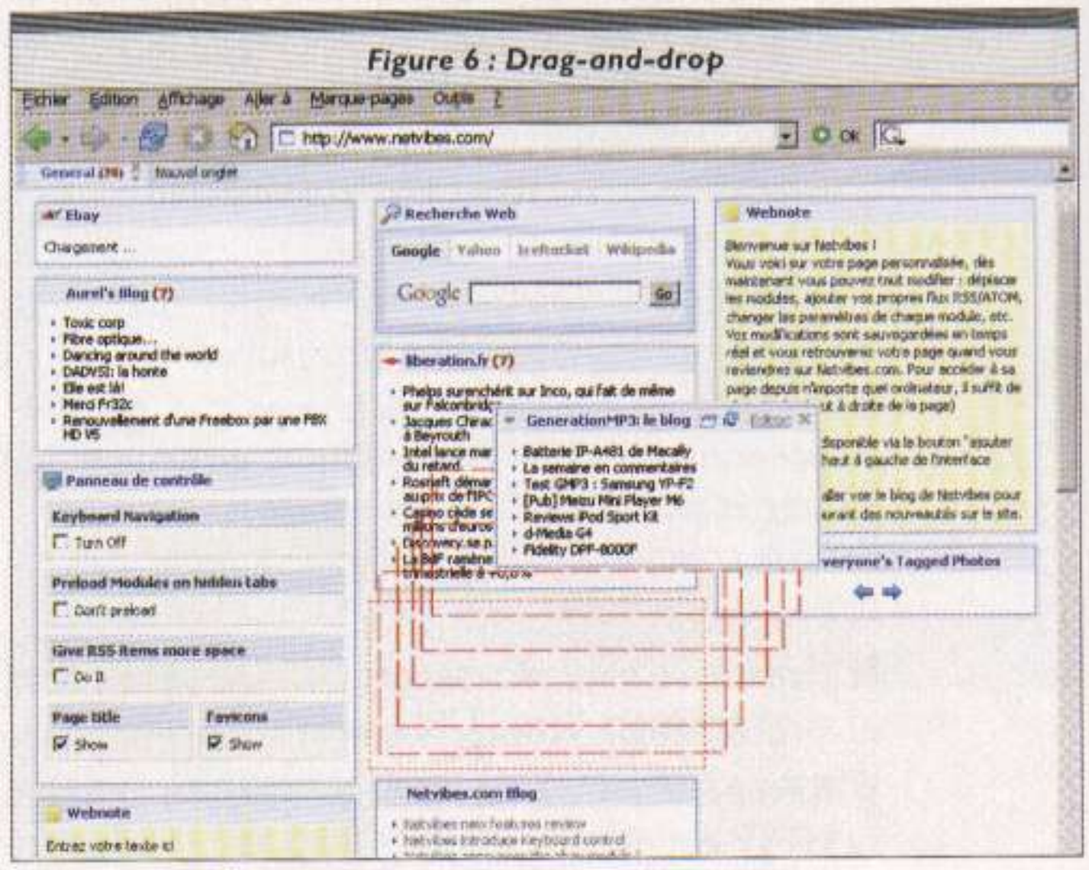
Un autre exemple consiste à alimenter des listes de sélections au fur et à mesure des choix de l'utilisateur. Une première liste propose les départements. Lorsque l'utilisateur a fait son choix, une requête AJAX permet d'obtenir tous les codes postaux du département. Lorsque l'utilisateur choisit un code postal, toutes les communes correspondantes sont proposées (figure 5). Sans AJAX, la page propose généralement un bouton permettant de faire une validation partielle du formulaire. Cela génère une nouvelle page complète, dont une liste de saisie complémentaire est alors initialisée. AJAX permet d'économiser ces échanges coûteux, tout en proposant une ergonomie plus cohérente.

Figure 5 : Listes déroulantes liées



Plus fort encore, des sites proposent de réorganiser les éléments d'un portail par drag-and-drop (figure 6).

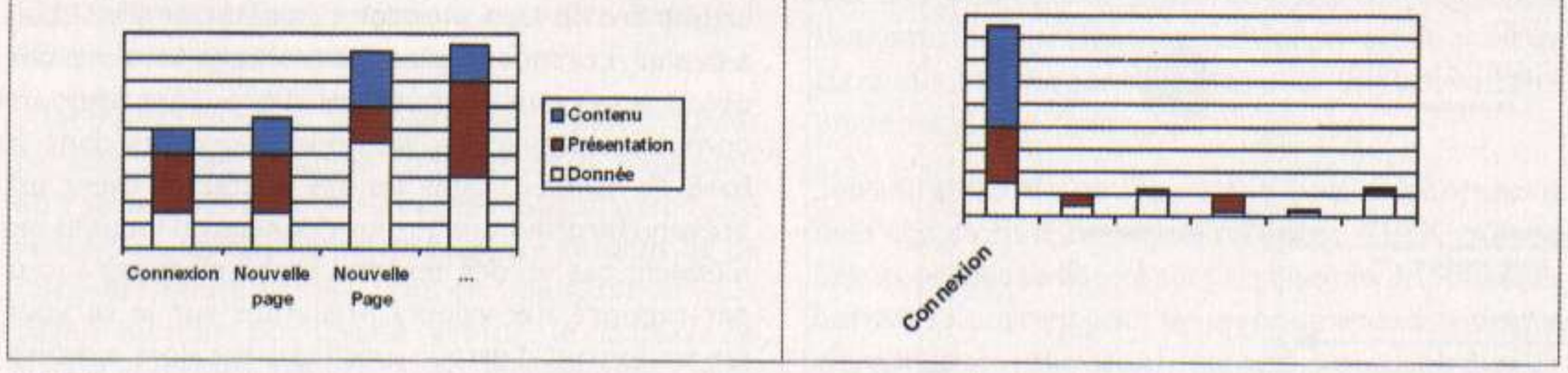
Figure 6 : Drag-and-drop



1.1 Optimisation du réseau ?

Si l'application AJAX est bien construite, elle permet de réduire fortement la bande passante (figure 7).

Figure 7 : Comparaison du trafic réseau



AJAX peut réduire le volume des informations échangées, mais peut aussi écrouler le serveur. En effet, si trop de requêtes arrivent sur le serveur, celui-ci peut être saturé. Pour pallier cet inconvénient, il peut être judicieux d'organiser les soumissions vers le serveur. Il n'est pas nécessaire de générer une requête AJAX pour chaque caractère ou à chaque modification d'un champ. Il est préférable d'agréger plusieurs requêtes clientes dans une seule requête HTTP. Les requêtes sont postées dans la boîte aux lettres. Celle-ci est relevée périodiquement pour être envoyée au serveur. Chaque requête peut être identifiée par un ID, afin de pouvoir être écrasée si nécessaire, avant la levé du facteur.

Dans l'exemple présenté figure 8, un utilisateur sélectionne un premier champ, tape sur la touche 'A', la touche 'B', puis sélectionne un deuxième champ et saisit '999'. Enfin, l'utilisateur corrige sa saisie pour la remplacer par '0'. Des validations sur le serveur sont souhaitées pour l'application. Dans le premier scénario, sans optimisation, pratiquement chaque événement génère une requête AJAX. Dans le deuxième scénario, les requêtes sont accumulées avant d'être envoyées au serveur. Avec cette nouvelle approche, il peut y avoir un délai supplémentaire avant que le serveur ne détecte une anomalie, mais le nombre de requêtes est réduit. Certaines requêtes ont même disparues. Par exemple, la saisie de la valeur '999' n'est pas signalée au serveur, car elle est immédiatement modifiée par la valeur '0'. La méthode `abort()` permet également d'interrompre une requête AJAX dont ont sait, un peu tard, que le résultat ne sera pas utilisé.

Avec une aide à la saisie, il y a un fort risque de saturer le serveur. Il faut impérativement faire une maquette et la tester en charge ! Ce n'est pas facile à cause des interactions utilisateurs complexes des applications AJAX. Les tests unitaires automatiques sont plus difficiles à réaliser avec une application AJAX qu'avec une application classique. Une capture du flux réseau pour pouvoir le rejouer permet d'identifier les problèmes d'infrastructures.

Le gain attendu en performance, en ne rafraîchissant pas toute la page, peut être annulé par une aide à la saisie. Le serveur et toute la chaîne de traitement doivent être adaptés pour pouvoir accepter de nombreuses petites requêtes.

1.2 Push : état du serveur

Pendant que l'utilisateur manipule son écran, des modifications peuvent intervenir sur le serveur. Pour connaître les modifications d'états du serveur, il y a deux approches :

- Interroger le serveur périodiquement **[9]** (Arrivée d'un nouveau mail ?).
- Désynchroniser l'invocation du serveur pour que celui-ci puisse prévenir le client immédiatement.

La deuxième approche est plus complexe, mais elle permet une réaction immédiate du client et réduit théoriquement le trafic réseau. Une requête est envoyée au serveur (figure 9). Celui-ci ne répond pas. Il laisse la requête en attente de réponse. Lorsque le serveur désire informer le client d'un événement, il répond enfin à la requête. Le client traite cela comme un événement du serveur et relance immédiatement une nouvelle requête en attente de l'événement suivant.

Figure 8 : Agrégation des requêtes

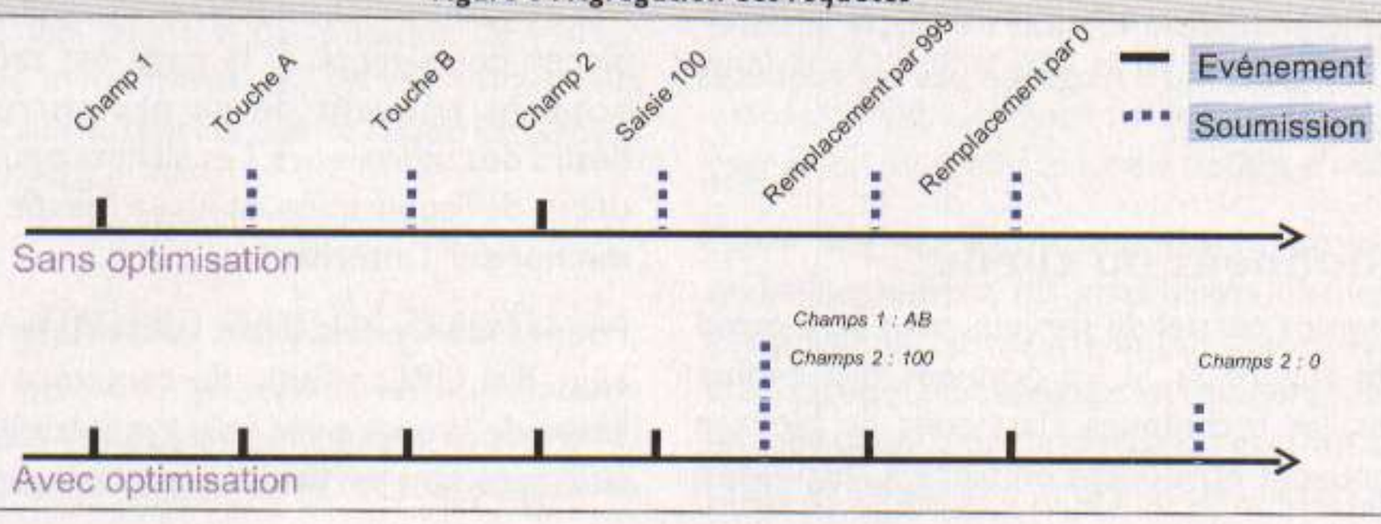
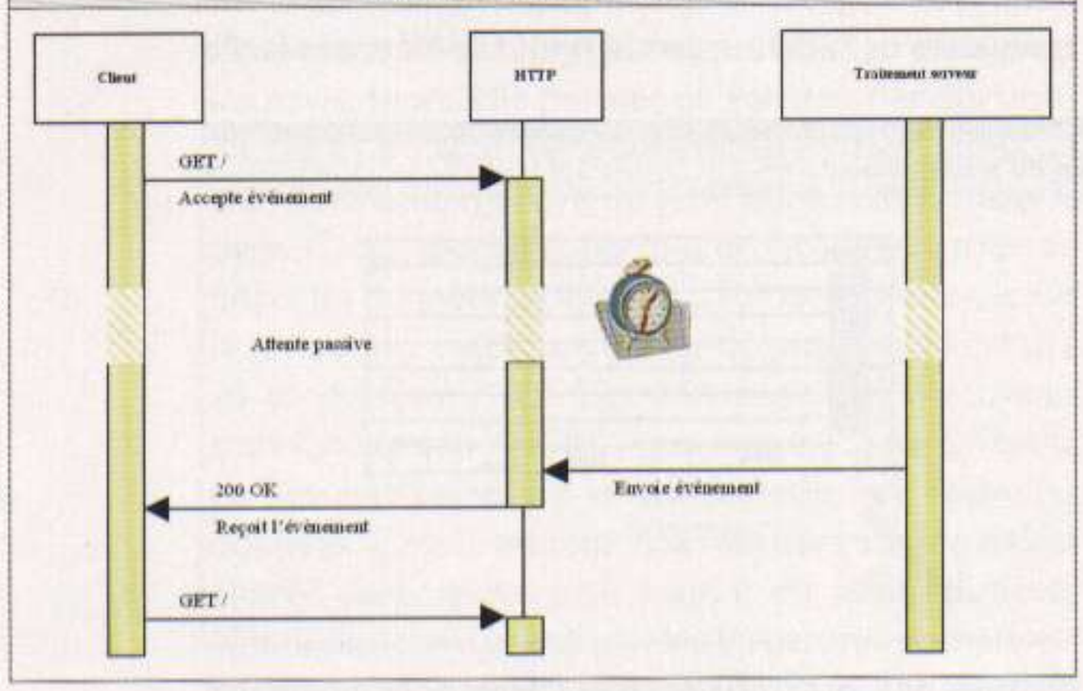


Figure 9 : Attente passive



Attention, cela risque de saturer le pool de connexion sur le serveur. En effet, les serveurs savent traiter un nombre limité de requêtes. En gardant des requêtes ouvertes, de nouvelles connexions risquent d'être refusées. Certains serveurs d'applications proposent des mécanismes spécifiques, afin de permettre une désynchronisation entre l'analyse de la requête et l'émission de la réponse. BEA, par exemple, propose une *servlet* spécifique `AsyncServlet`, permettant de ne pas exploser le serveur d'application.

En théorie, cela fonctionne. Mais, à cause des *proxies* et pare-feu, il est probable qu'une connexion, restée ouverte, soit automatiquement fermée après trente secondes. Cela oblige le client à rouvrir une connexion. Nous retombons alors dans la situation d'un pool dont la période est de trente secondes. Si la modification d'état du serveur doit être signalée le plus rapidement possible, situation d'un « chat » par exemple, il est judicieux d'utiliser cette technologie.

L'impact est parfois plus gênant si un proxy d'entreprise agrège les requêtes de différents clients dans la même connexion vers le serveur. Cette approche, autorisée par les proxies HTTP, permet de réduire le nombre de connexions vers le serveur cible. Le protocole HTTP 1.1 permet de cumuler les requêtes dans la même connexion. Les réponses doivent arriver dans l'ordre des requêtes. Cela entraîne qu'une requête n'est exécutée qu'après la précédente, car les réponses doivent arriver dans l'ordre. L'approche de Push consistant à garder la connexion ouverte, toutes les transactions suivantes sont bloquées. Il faut donc utiliser le protocole version 1.0 ou paramétrer le proxy HTTP pour qu'il n'agrège pas les requêtes d'utilisateurs différents.

1.3 Les données du client

Un état au moins partiel du serveur peut être gardé par le client (figure 3). Si les données ont évolué entre-temps, les techniques classiques de blocage optimiste peuvent être mises en place. C'est-à-dire que le client manipule des copies des données présentes sur le serveur. Entre-temps, elles peuvent évoluer sur le serveur, le client n'en est pas informé. Cette situation étant considérée comme rare, il n'est pas nécessaire de mettre en place des mécanismes complexes de blocage. À la place, un *timestamp* ou un numéro de version permet de détecter si un objet a évolué. Lorsque le client renvoie ses versions des objets, le serveur peut vérifier si elles sont toujours compatibles avec les versions courantes dans la base de donnée. Dans les cas limites, le client est prévenu tardivement que les données qu'il a utilisées n'étaient pas valides, que son cache n'est pas à jour par rapport aux valeurs présentes sur le serveur. Un traitement d'erreur spécifique est alors exécuté. L'utilisateur est informé du problème et doit prendre une décision : écraser les données dans la base de données ou rafraîchir les données sur le client et recommencer à y apporter les modifications. Cette situation étant rare, cette stratégie de blocage est qualifiée d'optimiste.

Où placer les données du client ? Elles peuvent être présentes dans la page si l'application propose l'approche « Une seule page pour l'application ». Si l'utilisateur peut naviguer entre plusieurs pages, les données sont perdues d'une page à une autre. Pour une durée de vie plus importante, il faut s'appuyer sur les caches HTTP, des IFrames, les cookies du navigateur (4 Ko x 20 cookies = 80 ko) ou les cookies « Flash » qui acceptent une taille plus importante. Avec des privilèges, il est possible d'utiliser le disque dur de l'utilisateur (via un ActiveX ou une *applet* Java signée).

Attention, aucune donnée présente chez le client n'est digne de confiance. En effet, celui-ci peut les manipuler avant de les envoyer vers le serveur. De même, un virus ou une attaque de type *Cross Site Scripting* (injection de Javascript dans une page) peut révéler à un pirate des informations confidentielles. Il faut se prémunir contre cela en cryptant éventuellement les données persistantes chez le client et en auditant le site pour éviter les injections de script.

1.4 Gestion de la navigation

Pour ne pas perdre les utilisateurs avec une nouvelle ergonomie, il ne faut pas modifier leurs habitudes. Les internautes utilisent les boutons « suivant », « précédant », « rafraîchir » et souhaitent pouvoir placer des signets. Si la page est très mobile, ces boutons risquent de ne pas correspondre aux désirs des utilisateurs. Les signets peuvent mener au début de l'application, et non à l'étape que souhaitait mémoriser l'internaute.

Pour résoudre cela, il faut utiliser les marqueurs intra page des URL, à l'aide du caractère dièse. Chaque étape de la page peut être synthétisée par une URL différente, sans rafraîchissement de la page. L'instruction

`window.location.replace(newHash)` permet de faire évoluer l'URL de la page. Par exemple, l'URL peut devenir quelque chose comme ceci : **http://monsite.org/ajaxapp#wizard:step2**. L'utilisateur peut alors mémoriser un signet pour l'étape deux du Wizard. Lors de l'initialisation de la page, l'URL est analysée. Cela permet à l'application de restituer la page à l'étape correspondante.

Si les modifications ne sont pas facilement synthétisées par une simple valeur, le contexte de la page peut être sérialisé et encodé en Base64 (encodage binaire vers ASCII, utilisant 64 symboles). Le résultat de ce calcul est ajouté à l'URL. Lors de l'initialisation de la page, il est alors possible de restituer le contexte de la page et d'ajuster l'affichage en conséquence. Cette approche consiste à avoir une sorte de session sur le client, session mémorisée dans l'URL ! Des frameworks proposent des services **[10]** de manipulation d'URL pour les applications AJAX.

1.5 Gestion des erreurs

À l'usage, la gestion des erreurs est complexe. Les requêtes étant asynchrones, les erreurs sont découvertes tardivement. Entre-temps, l'utilisateur est déjà en train de saisir un autre champ. Il faut impérativement mettre en place un mécanisme de notification de l'utilisateur ne perturbant pas sa saisie. Par exemple, l'ajout d'un message d'erreur à côté du champ ; le changement de couleur du champ ou du libellé ; l'ajout d'icône d'état de tous les traitements asynchrones. Il peut être nécessaire de classifier les erreurs pour adapter la notification de l'utilisateur suivant l'urgence. Sinon, l'utilisateur risque de recevoir une alerte concernant un champ saisi, il y a quelque temps. L'utilisateur est mentalement parti vers d'autres problèmes. L'alerte va lui faire interrompre la saisie en cours, générant plus tard une autre alerte, etc.

Une approche pour faire cela consiste à utiliser le marqueur `<label/>` qui permet d'associer un texte à un champ, et permet aux outils de synthèse vocale d'informer les malvoyants du libellé de la saisie.

```
<label for="nom">Nom</label> <input size="20" type="text" name="nom" />
```

Un framework peut alors retrouver automatiquement le libellé et lui modifier son style CSS.

Les traitements AJAX étant essentiellement asynchrones, cette particularité doit être prise en compte lors de la conception des pages. Il est possible de rendre synchrone les invocations du serveur, mais cela s'effectue avec une attente active et risque fortement de figer le navigateur.

2. Pour quelles applications ?

Alan Cooper propose de classer les applications en deux familles **[11]** : les applications souveraines et les applications transitoires. Les applications souveraines sont utilisées quotidiennement pour de longues périodes. Elles prennent toute l'attention de l'utilisateur. Elles sont conçues pour une utilisation optimum, au prix d'une formation.

Les applications transitoires sont utilisées occasionnellement pour de courtes périodes. L'utilisateur interrompt son activité principale avant de la reprendre. Elles sont conçues pour une accessibilité immédiate, sans formation.

Suivant la famille de l'application, les améliorations de l'ergonomie seront différentes. AJAX offrant beaucoup plus de liberté sur l'ergonomie, de nouveaux horizons s'ouvrent pour les ergonomes. Les principes d'ergonomies nouveaux (drag-and-drop, contrôles en tâche de fond, etc.) peuvent dérouter les utilisateurs. Avant AJAX, les ergonomies possibles pour proposer un processus étaient évidentes, imposées par les contraintes du HTML. Dorénavant, AJAX permet plus d'ouverture et d'interaction avec les utilisateurs. Les internautes ont pris des habitudes leur permettant d'être immédiatement efficaces sur n'importe quel site Internet. Apporter des innovations bousculant leurs habitudes risque d'être contre productif.

Pour les applications transitoires, AJAX doit être utilisé avec parcimonie, afin d'aider l'utilisateur sans trop le bousculer. L'aider et le conseiller lors de la saisie d'un critère de recherche est une bonne chose. Les listes déroulantes liées permettent d'éviter les soumissions partielles d'un formulaire et améliore discrètement l'ergonomie, tout en restant dans les usages du net.

Par contre, l'utilisation du drag-and-drop n'est pas évidente pour un internaute arrivant pour la première fois sur un site. De même, sauver les modifications de l'utilisateur dès qu'il coche une case à cocher n'est pas conforme aux usages. L'utilisateur va chercher en vain le bouton de soumission. Cacher ou présenter des zones sur une page ou proposer des onglets ne perturbe pas l'utilisateur.

Que les informations complémentaires soient chargées en tâche de fond et non lors de la récupération de la page ne concerne pas l'internaute ! Il s'agit de problèmes techniques, résolus par AJAX, à l'insu de l'utilisateur. Il faut éviter d'aller trop loin et de devoir former l'utilisateur à l'utilisation du site. Il ne faut pas oublier que le concurrent est à un clic de là ! Ces technologies doivent être utilisées après une bonne analyse du public visé et de l'investissement qui lui est demandé. Les applications transitoires peuvent gagner en attractivité et en facilité d'utilisation.

Pour les applications souveraines, utilisées quotidiennement, les possibilités d'amélioration de l'ergonomie sont plus importantes. Cela permet d'améliorer l'efficacité des utilisateurs. Des tests **[12]** en laboratoire ont montré un gain de 30% pour la saisie et l'édition d'informations comptables.

AJAX peut également être utilisé pour enrichir les portails. Une page d'un portail est décomposée en plusieurs éléments graphiques, souvent autonomes les uns les autres. AJAX offre la possibilité de faire évoluer chaque *portlet* sans devoir imposer un rechargement de toute la page. La personnalisation du portail peut également bénéficier des techniques de drag-and-drop. Les portlets peuvent être réorganisés physiquement sur la page, sans devoir proposer une ergonomie complexe (figure 6). Il existe des Best Practices « JSR168 & AJAX **[13]** ». Le JSR286 (Portlet 2.0) y réfléchit également.

Les projets AJAX entraînent une réelle réflexion sur l'ergonomie. Il est facile de louper la cible, en proposant une ergonomie élégante et animée, mais moins efficace qu'une application WEB classique.

3. Les pièges

AJAX présente un certain nombre de pièges qu'il est bon de connaître avant de se précipiter à concevoir une nouvelle application. Il faut déjà savoir qu'une application AJAX peut réduire l'accès à l'application à une partie de la population. Par exemple, les utilisateurs handicapés **[14]** ne peuvent plus y accéder avec leurs extensions habituelles (synthétiseur vocaux, MozBraille **[15]**, etc.). Il existe des règles d'accessibilité **[16]** imposées par les administrations. Les applications AJAX sont incompatibles avec les règles d'accès aux contenus Web **[17]**. Le W3C propose une feuille de route **[18]** pour offrir des interfaces riches à cette population.

Les navigateurs anciens, ou dont les règles de sécurité interdisent l'utilisation de Javascript, ne peuvent plus utiliser l'application. Il faut prévoir une solution en mode « dégradé ». Gmail, par exemple, propose deux versions de l'application. L'une utilise AJAX, l'autre n'utilise ni AJAX ni le Javascript.

Si AJAX est utilisé pour améliorer l'ergonomie, il est relativement facile d'avoir une stratégie de repli plus classique. Par exemple, le champ de saisie d'un moteur de recherche ne bénéficie plus alors des suggestions ; les listes déroulantes liées rechargent toute la page dès la perte du *focus* au lieu de se contenter d'obtenir les quelques informations manquantes ; des boutons ou des formulaires compensent l'absence de drag-and-drop d'un portail. Pour les applications transitoires, AJAX doit être utilisé comme un complément à une application Web classique, et non comme un élément majeur de l'ergonomie.

Les pages AJAX doivent pouvoir être indexées par les moteurs de recherches, sous peine de disparaître du net. Si l'application ne propose pas une ergonomie du type « application dans une seule page », les moteurs continueront à pouvoir référencer les pages. Sinon, il faut proposer des pages particulières pour les robots. Lors de la détection du robot de Google ou d'autres, des pages non-AJAX sont retournées, avec les données importantes.

Les développeurs doivent être vigilants sur la taille du code Javascript nécessaire à l'application. Même compactés **[19]**, les Javascripts AJAX nécessaires peuvent être très volumineux (>100 ko). Il faut compenser cela avec des paramètres de cache de la requête HTTP, et une réutilisation sur de nombreuses pages. Ils doivent également faire attention à la quantité de mémoire nécessaire sur le client. Dès la conception, il faut chercher à économiser les ressources. Tous les internautes n'ont pas la chance d'avoir un terminal généreusement pourvu. Des frameworks éviteront de consommer quatre-vingt pourcents du temps de développement à régler les problèmes de portabilité.

Les requêtes AJAX ne maintiennent pas toujours en vie la session **[20]** de l'utilisateur. Afin de ne pas la perdre lors de l'utilisation d'une application AJAX, il faut générer une requête périodiquement (entraînant un trafic supplémentaire) ou que l'application AJAX demande l'état de la session avant de laisser l'utilisateur continuer à manipuler les données. La durée de vie de la session d'une application AJAX peut également être adaptée au contexte. Ces problèmes disparaissent si l'intégration d'AJAX est superficielle, et que les pages HTML sont toujours utilisées, mais enrichies.

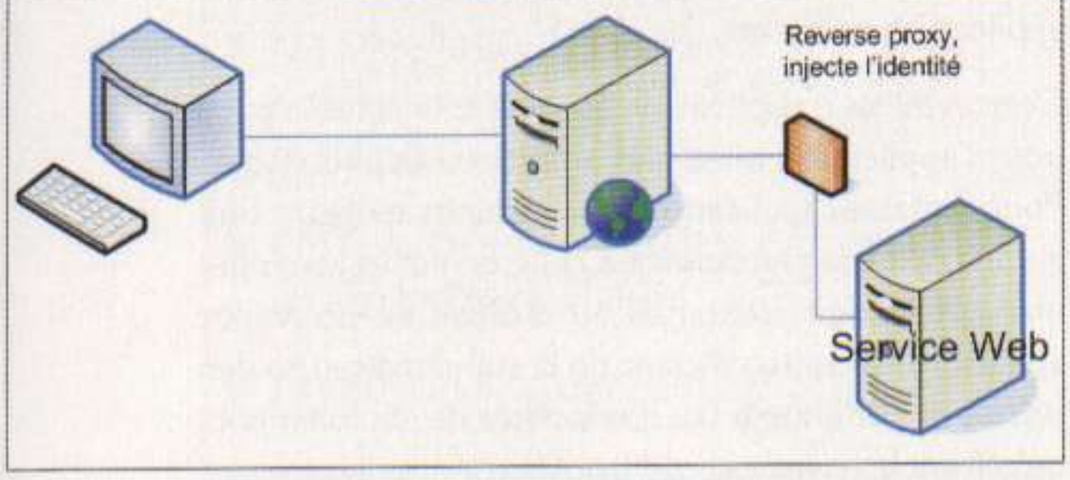
Internet Explorer ne possède pas de ramasse-miettes efficace pour le code Javascript **[21]**. Cela ne posait pas de problème pour les applications changeant régulièrement de page, car IE fait alors le ménage. Les applications AJAX sont différentes. Une page peut être utilisée pendant longtemps, générant des fuites mémoires très difficiles à localiser.

Les impressions et l'utilisation « *off-line* » ne sont pas traitées par AJAX. Il est également difficile de proposer une approche multi-canal type PDA, WebTV, etc. L'expérience sur de nombreux sites a permis d'identifier quelques cent cinquante-trois règles de bonnes pratiques pour les services en lignes **[22]**. Les applications AJAX en violent plusieurs.

Au niveau sécurité, des difficultés nouvelles peuvent arriver. Les navigateurs interdisent au client de communiquer avec un serveur différent de celui ayant livré le script. Internet Explorer affiche alors une fenêtre demandant l'autorisation à l'utilisateur. Celle-ci n'est affichée qu'une seule fois. Avec Mozilla Firefox, une fenêtre est affichée pour chaque requête. Cela peut être levé en signant les scripts. Pour contourner cette difficulté, il faut proposer un *reverse proxy* **[23]** sur le serveur, pour pouvoir invoquer d'autres sites. Cela peut présenter un risque de vulnérabilité et intéressera sûrement les pirates en mal d'anonymat.

Un reverse proxy de service Web (figure 10) peut ajouter les paramètres d'identifications à la requête, à partir des informations de la session d'utilisateur. Cela permet de propager l'identité de l'utilisateur à l'aide du serveur, sans présenter de vulnérabilité sur le client. Un ESB (*Entreprise Service Bus*) peut jouer ce rôle d'enrichissement du service Web.

Figure 10 : Reverse proxy



Les applications AJAX peuvent également présenter des risques de sécurité. Par exemple, les cookies mémorisant l'état du client sont visibles et manipulables. Le serveur expose des services métier. Ceux-ci doivent être codés en intégrant le risque d'être invoqué en dehors de l'application. Les échanges avec le serveur doivent être sécurisés, cryptés et authentifiés, les paramètres vérifiés.

Les frameworks AJAX peuvent présenter des risques de déni de service ou des vulnérabilités pour le serveur. Si le source n'est pas public, l'audit n'est pas toujours possible. Par exemple, un audit rapide du framework de Google (GWT **[24]**) montre qu'il est possible de consommer toute la mémoire du serveur en envoyant un flux bien choisi. L'algorithme de dé-sérialisation ne possède pas de limite pour la taille des tableaux en mémoire. En envoyant un flux indiquant un tableau d'octets d'un million d'éléments, le serveur va allouer l'espace nécessaire avant de lire chaque octet.w

Un code AJAX mal contrôlé peut envoyer à son propre serveur des requêtes à l'infini, créant ainsi une auto-attaque par saturation. Il existe des virus AJAX **[25]** et des outils **[26]** d'analyse de code AJAX.

Plus de 150 frameworks sont recensés sur **Ajaxpatterns.org**. C'est un monde très instable, car les pré-requis techniques sont simples. Il suffit de savoir utiliser l'objet `XMLHttpRequest`. Certains frameworks se détachent et sont soutenus par des entreprises renommées. Des produits commerciaux proposent des approches originales et riches.

Les approches sont très différentes. Certains ne s'occupent que du tiers client, d'autres facilitent la communication entre le client et le serveur, suppriment l'adhérence à Javascript ou enrichissent des frameworks existants. Certains sont orientés « une seule page par application » ou proposent d'enrichir le XHTML de comportements dynamiques. L'intégration d'AJAX s'effectue alors sans aucun Javascript.

A mon avis, l'approche la plus prometteuse consiste à enrichir les frameworks serveur, orientés « composant ». C'est le cas de JSF et de .Net. Ces frameworks proposent de réutiliser et d'assembler des composants d'ergonomies. Sun vient de publier un tutoriel **[27]** pour l'intégration d'AJAX dans JSF.

Ces frameworks gardent une connaissance fine de la structure de la page. Si un événement client peut avoir un impact sur un composant coté serveur, une requête est envoyée et l'information est propagée au composant. La page est alors entièrement recalculée avant d'être retournée au client.

En greffant des composants AJAX, la communication entre le composant présent dans le client et son alter ego sur le serveur s'effectue en direct. Il n'est plus nécessaire de recalculer toute la page. De plus, les composants peuvent théoriquement proposer deux versions : une classique sans utilisation d'AJAX et une version utilisant AJAX. Ainsi, il n'est plus nécessaire de rédiger deux fois la même application. Si le navigateur refuse les fonctionnalités AJAX, l'application s'adapte. Pour la saisie d'une ville par exemple, le composant peut proposer une aide à la saisie ou un bouton de validation partielle du formulaire ; les listes déroulantes liées peuvent être traduites comme un événement JSF ou .NET classique, ou transformées en événement AJAX.

Coté client, je préconise d'utiliser des frameworks proposant un moteur générique de transformation des données pour les insérer dans la présentation. Certains utilisent des annotations du XHTML, d'autres des feuilles de styles, etc. Ces approches permettent de s'affranchir des problèmes de Javascript.

Il y a une évolution dans le rôle du navigateur et du serveur. Avant, le navigateur présente un contenu ; le serveur héberge une application ; les requêtes au serveur sont explicites. Après, le navigateur héberge une application ; le serveur fournit des données ; la plupart des requêtes au serveur sont implicites.

4. Conseils

Un seul objectif : générer de la valeur en améliorant la productivité et/ou l'attractivité. Il faut utiliser AJAX à bon escient :

- Sur les champs de moteur de recherche.
- Sur les pages mettant en place un processus à plusieurs étapes.
- Pour manipuler de longues listes de données.
- Pour réduire le risque d'une mauvaise saisie de l'utilisateur ou pour effectuer des vérifications de surface. Attention à ne pas effectuer cela sur chaque champ.
- N'utiliser les fonctions Drag-and-drop que si cela apporte un véritable avantage ergonomique. Les réserver de préférence aux applications souveraines.

Limiter AJAX au nécessaire. S'il s'agit d'améliorer la productivité, on se concentrera sur les écrans ou fonctions les plus utilisées. Il faut éviter de casser les conventions (retour, signet, indexation par les moteurs de recherches, etc.) et indiquer les changements **[28]** sur la page ; expliquez à l'utilisateur par un message temporaire pourquoi il attend.

Proposez un framework de notification non intrusif sur l'interaction de l'utilisateur. Si une information est à communiquer au client, lui indiquer directement dans la page et non par une boîte d'alerte.

Pour ne pas s'exclure d'une partie du public, il faut proposer des alternatives à l'utilisation d'AJAX.

Pour coder l'application, ne réinventez pas la roue : utilisez des frameworks après les avoir sélectionnés sur la pérennité des stratégies mises en place. Les choix seront différents s'il s'agit d'intégrer AJAX à des applications existantes ou pour de nouvelles applications. Commencez par normaliser vos pages en XHTML. Ce format rigoureux facilite ensuite les manipulations.

Coté serveur, il faut répondre aux requêtes le plus vite possible. Utilisez des caches puissants pour éviter d'interroger trop souvent la base de donnée. Paramétrez les requêtes HTTP pour exploiter toutes les possibilités de caches du protocole (En-tête ETAG **[29]**). Pensez au maintien en vie de la session de l'utilisateur.

Coté client, utilisez un cache Javascript sur le navigateur pour réduire le nombre de requêtes vers le serveur (agrégation de requêtes, etc.). Utilisez les *Design Patterns* **[30]** de développement en Javascript. N'oubliez pas de tester l'application sur tous les navigateurs et toutes les plateformes. Vérifiez son comportement sur un LAN et un WAN.

Conclusion

Il est raisonnable d'utiliser cette technologie en tant que complément à une application WEB classique. Offrir des aides à l'utilisateur, des suggestions de réponses, le remplissage dynamique des listes déroulantes ne perturbe pas les applications classiques. Il s'agit d'offrir des composants ergonomiques avancés. Avec cette approche, il est facile de proposer des solutions alternatives plus classiques, pour permettre à tous les internautes d'utiliser l'application. Cette approche évite de devoir gérer la durée de vie de la session, la gestion de la persistance des objets métier sur le client, une gestion complexe de l'historique ou de la navigation, d'exposer trop ouvertement les objets métier présents sur le serveur, etc. Dans le cas d'une nouvelle application, il est préférable d'utiliser un framework à base de composants (JSF ou .NET) pour limiter la complexité du client, et garder la maîtrise de l'utilisateur sur le serveur. Sinon, différents frameworks ou composants client permettent d'enrichir une application existante.

Concevoir une application du type « une seule page pour l'application » est plus ambitieux et plus risqué. Pour certaines applications souveraines, exigeant une grande richesse ergonomique, cela permet de *revamper* une application existante ou d'offrir de nouvelles applications, en bénéficiant de la standardisation des postes des utilisateurs et des facilités de déploiements qu'offrent les navigateurs. Mais il faut alors s'interroger si une approche RDA (*Rich Desktop Application*) n'est pas préférable. Il est vrai que le déploiement est un peu plus complexe qu'avec un navigateur et que les applications exigent plus de ressource sur le poste du client. Les technologies Java Web Start **[31]** de Sun ou No-Touch Deployment **[32]** de Microsoft offrent des approches séduisantes. Une application RDA sera plus à même de gérer les impressions, les interactions avec les autres applications ou les périphériques du poste de l'utilisateur (Office, carte à puce, lecteur de code barre, etc.)

Open Ajax Aliance **[33]** est une initiative qui se propose d'unifier les efforts sur AJAX. Il est trop tôt pour savoir s'il en sortira quelque chose.

Le phénomène Web 2.0 (usages et combinaison des technologies Web telle AJAX) est un axe de recherche fort pour les équipes Innovation d'Atos Origin. L'objectif est d'accompagner nos clients afin de tirer profit de l'usage de ces nouvelles technologies, ainsi que pour la création de nouveaux services. Nous pensons qu'AJAX peut apporter de réelles opportunités d'attractivité et d'efficacité pour les utilisateurs. Les projets réussis utilisent AJAX raisonnablement et offrent des solutions alternatives pour les internautes ne pouvant bénéficier de cette évolution technologique. L'utilisation d'un framework est indispensable.

Philippe Prados,

Architecte chez Atos Origin

www.prados.fr

RÉFÉRENCES

- **[1] http://en.wikipedia.org/wiki/Jesse_James_Garrett**
- **[2] http://www.adaptivepath.com/publications/essays/archives/000385.php**
- **[3] http://www.thecounter.com/stats/2006/July/browser.php**
- **[4] http://www.json.org/java/**

- [5] http://ws.apache.org/axis/
- [6] http://www-128.ibm.com/developerworks/web/library/ws-wsajax/?ca=dgr-lnxw07ISOAP-Ajax
- [7] http://www.codeproject.com/soap/JavaScriptProxy_01.asp
- [8] http://www.openrico.org/
- [9] https://bpcatalog.dev.java.net/nonav/ajax/refreshing-data/design.html
- [10] http://codinginparadise.org/projects/dhtml_history/README.html
- [11] http://www.cooper.com/articles/art_your_programs_posture.htm
- [12] http://www.developer.com/java/other/article.php/3554271
- [13] http://developers.sun.com/prodtech/portalserver/reference/techart/ajax-portlets.html
- [14] http://www.acces-pour-tous.net/
- [15] http://mozbraille.mozdev.org/
- [16] http://www.adae.gouv.fr/article.php3?id_article=246
- [17] http://www.w3.org/TR/WAI-WEBCONTENT/
- [18] http://www.w3.org/TR/2006/WD-aria-roadmap-20060926/
- [19] http://www.google.fr/search?q=javascript+compress
- [20] http://forum.alsacreations.com/topic-24-10462-1-AJAX-et-sessions.html
- [21] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/IETechCol/dnwebgen/ie_leak_patterns.asp
- [22] http://www.opquast.com/bonnes-pratiques/liste/
- [23] http://developer.yahoo.com/javascript/howto-proxy.html
- [24] http://code.google.com/webtoolkit/
- [25] http://namb.la/popular/tech.html
- [26] http://www.denimgroup.com/Sprajax/
- [27] http://java.sun.com/javaee/javaserverfaces/ajax/tutorial.jsp
- [28] http://www.yourtotalsite.com/archives/javascript/yellowfade_technique_for/Default.aspx
- [29] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.19
- [30] http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
- [31] http://java.sun.com/products/javawebstart/
- [32] http://support.microsoft.com/kb/837909/en-us
- [33] http://www.zimbra.com/blog/archives/2006/05/openajax_update.html

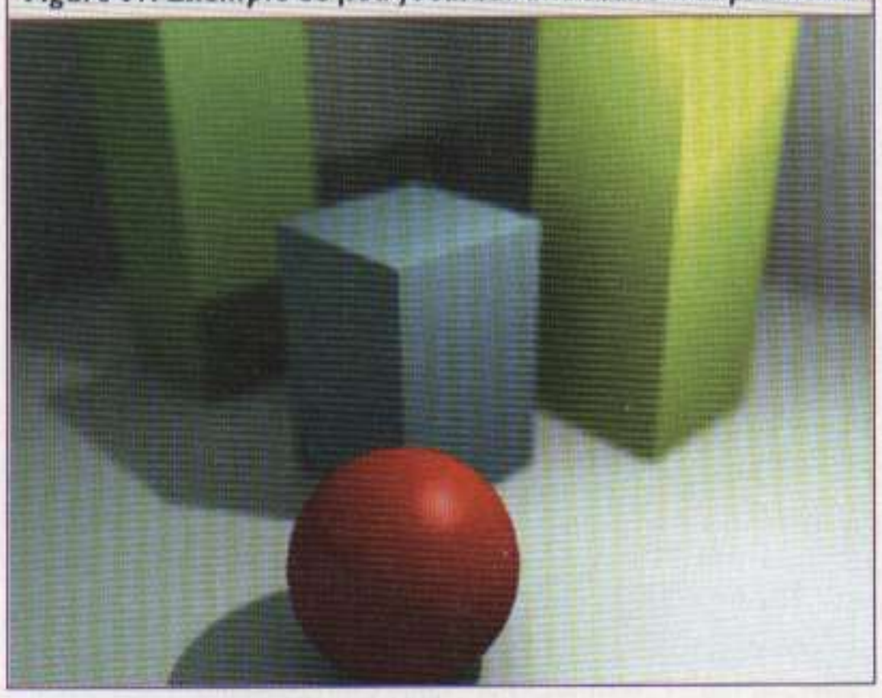
Olivier Saraja

Yafray, le moteur de rendu photoréaliste libre – maîtriser le flou focal

L'effet de flou focal est omniprésent dans les médias audiovisuels. Il permet de mettre en évidence des sujets importants dans les scènes, qu'elles soient photographiques ou cinématographiques, ou simplement dans un but artistique.

Vous ne savez pas ce qu'est le flou focal ? Vous avez certainement vu des scènes au cinéma ou à la télévision où un personnage du premier plan discute avec un autre personnage en arrière-plan. Celui qui parle est parfaitement discerné par les spectateurs, tandis que l'auditeur en retrait baigne dans le flou, et réciproquement lorsque celui qui a la parole change. C'est tout simplement de cela qu'il s'agit : donner à l'image une importance particulière à un objet ou une personne, au détriment du reste de la scène qui devient flou. De même, lorsque vous prenez des photographies de trop près, votre sujet sera certainement flou, sauf si vous disposez d'un autofocus ou d'un mode macro, ou que vous soyez suffisamment averti pour régler la focale de votre appareil sans l'aide de l'électronique (la plupart des appareils photo numériques modernes disposent en effet d'un autofocus).

Figure 01: Exemple de flou focal sur une scène très primitive



Dans leur quête de photoréalisme, les moteurs de rendu s'exercent tous plus ou moins à cet art difficile qu'est la simulation de l'effet de flou focal. On parle indifféremment de flou focal (*focal blur*) ou de champ de profondeur (*depth of field*). Dans sa première partie, cet article va explorer deux méthodes très classiques de simulation de cet effet, puis, dans sa seconde partie, il s'attachera à expliquer comment le mettre en œuvre avec Yafray.

Notions fondamentales (et rapprochement avec le jargon de nos amis photographes) :

Le point focal : c'est le point virtuel de l'espace sur lequel la caméra est pointée. Activement observé, il sera parfaitement net, du point de vue de la caméra. En photographie, on parle souvent de distance de mise au point (à partir de la caméra et dans la direction d'observation de celle-ci).

L'ouverture : terme directement emprunté au monde des photographes, l'ouverture (du diaphragme) permet de quantifier le niveau de flou d'une image. Plus le diaphragme est ouvert, plus l'image résultante sera floue. *A contrario*, plus il est fermé, plus l'image sera nette.

Le champ de profondeur : encore un terme directement emprunté au monde de la photographie, et qui découle directement des deux notions présentées ci-avant. Chez nos amis photographes, la profondeur de champ définit la zone de l'espace dans laquelle doit se trouver le sujet à capturer pour que l'on puisse en obtenir une image considérée comme nette. Dans notre monde virtuel, c'est l'ouverture qui définit l'espace net encadrant le point focal dans la direction d'observation.

1. Le flou focal : les méthodes

Il existe en particulier deux méthodes très conventionnelles pour simuler un flou focal, directement en cours de rendu. Il existe d'autres méthodes que celles présentées ici, en particulier, l'usage d'un logiciel de retouche d'image comme The Gimp, où il s'agit, par le biais de l'usage de calques ou de sélections, de n'appliquer un flou gaussien que sur une partie de l'image.

Tout au long de cet article, nous allons travailler sur la base de la scène d'exemple qui suit, créée et rendue dans Blender. Elle nous servira également à montrer les différences entre les deux méthodes conventionnelles de flou focal. On y parlera de point focal, qui spécifie le point virtuel qui sera toujours parfaitement net sur l'image rendue. Habituellement, il est défini par une distance spécifique depuis la caméra, et dans la direction visée par celle-ci, mais cela peut varier d'un logiciel à l'autre, ou d'un moteur de rendu à l'autre.

Figure 02 : Matérialisation du point focal dans Blender : une croix jaune sur la ligne de visée de la caméra indique sans ambiguïté la distance à laquelle le rendu sera parfaitement net

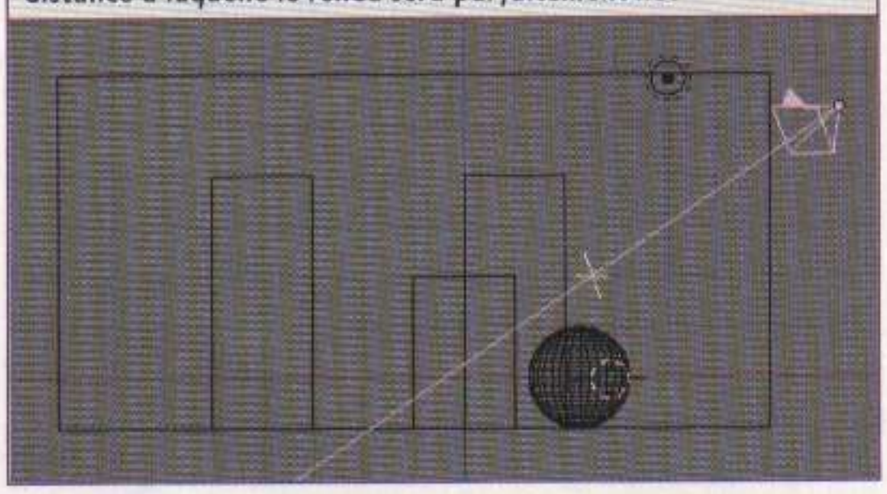
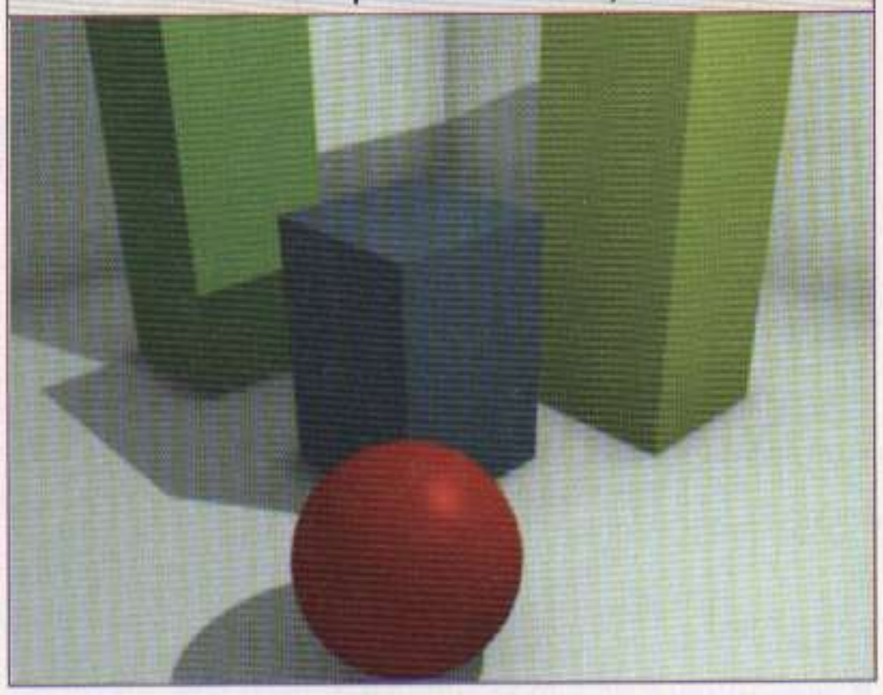


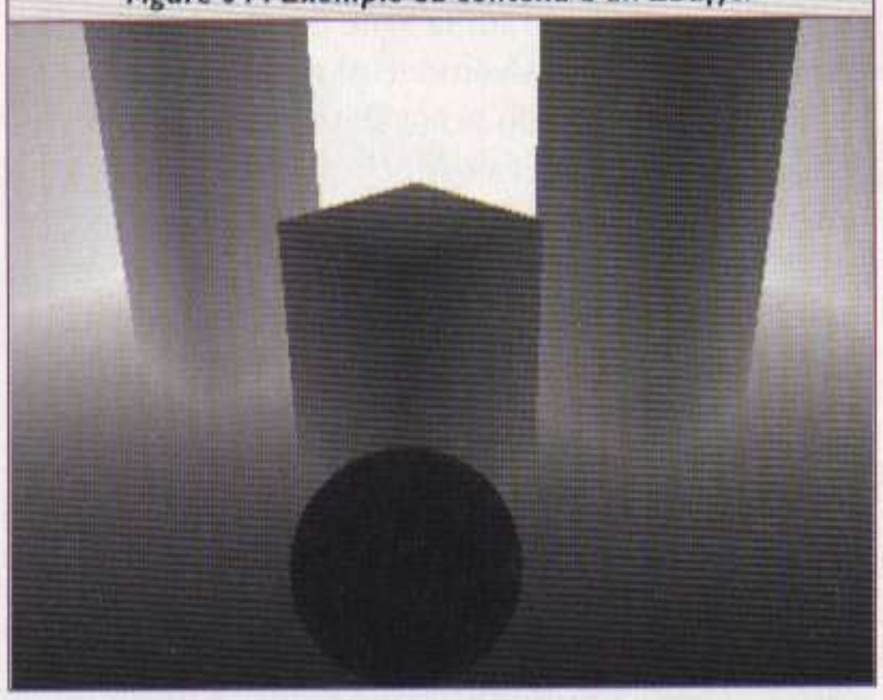
Figure 03 : La scène de base dans Blender (rendu effectué avec option de Radiosité)



1.1 Méthode 2D : usage du Zbuffer

Dans le cadre de cette première méthode, nous allons attribuer à chaque pixel de l'image une couleur qui ne sera pas propre à la couleur de l'objet rendu, mais un dégradé de gris, fonction de la distance qui sépare le pixel par rapport au point focal théorique. Par exemple, les pixels situés à la même distance que le point focal seront d'un noir parfait, et les points les plus éloignés totalement blancs. On appelle cela le tampon de profondeur Z, ou encore le Zbuffer.

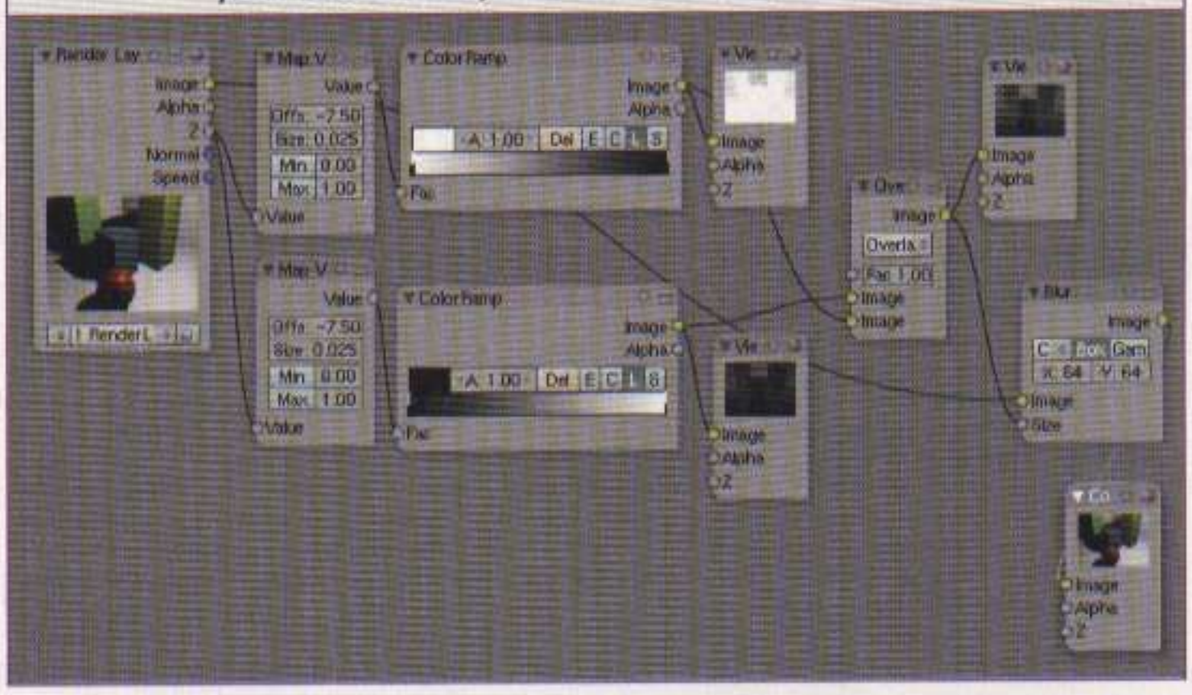
Figure 04 : Exemple du contenu d'un Zbuffer



L'usage de cette carte de profondeur est alors assez simple : on considère que les points totalement noirs sont totalement nets, et que les points totalement blancs sont totalement flous (intensité de flou maximale). Les points intermédiaires sont plus (clairs) ou moins (sombres) flous en fonction de leur position sur le dégradé de gris. Un simple filtre 2D permet alors de « flouter » chaque point en fonction de sa distance par rapport au point focal.

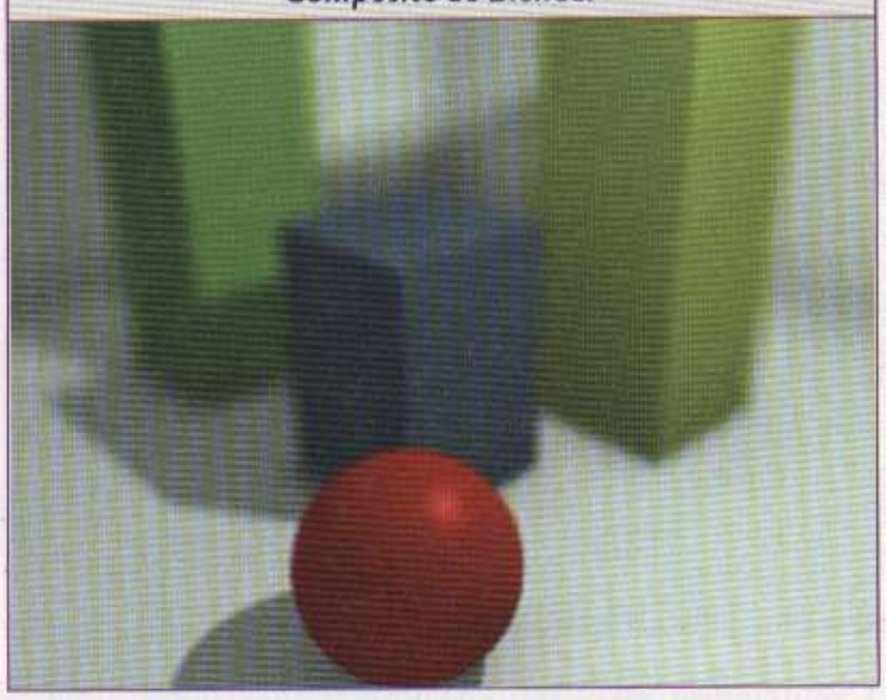
Un exemple d'application de cette méthode est celle du moteur de rendu interne de Blender. Grâce aux nœuds composites introduits dans Blender 2.42, il est possible de définir un dégradé de gris personnalisé qui permet de tenir compte de la position d'un point focal qui n'est autrement pas défini, grâce aux nœuds *Map Values* (paramètre *Offset*). Une fois la carte de floutage obtenue par superposition (méthode *Overlay* du nœud *Mix*) du dégradé devant le point focal et celui derrière le point focal, un nœud *Blur* permet de flouter l'image d'origine en fonction de l'intensité des gris ; le nombre d'échantillons utilisés lors de l'opération sont définis par les paramètres *X* et *Y* du nœud *Blur*, ainsi que l'algorithme de flou (*CatRom*, dans le même nœud).

Figure 05 : Exemple d'un réseau nodal permettant de simuler un effet de flou focal dans Blender, avec son moteur de rendu interne



Le résultat est plus ou moins convaincant, suivant l'importance attachée par l'exploration de l'utilisateur des paramètres *Offset* et *Size* du nœud *Map Values*, ainsi que des algorithmes, l'usage de *bokehs* ou encore le nombre d'échantillons de flou dans le nœud *Blur*.

Figure 06 : Exemple de flou focal grâce aux nœuds Composite de Blender

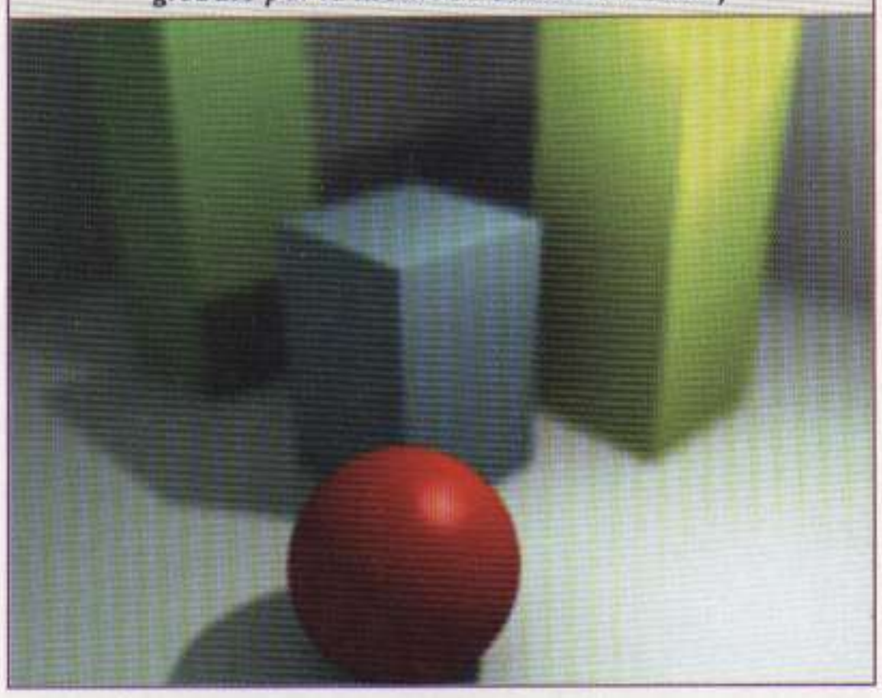


Cette méthode présente quelques avantages certains sur la version *raytracée* du flou focal : s'agissant d'un post-traitement 2D à une image rendue, l'image finale est très rapide à calculer, et s'en trouve particulièrement utile dans le cadre du rendu d'animations. Le réseau nodal, en revanche, demande un minimum d'expertise de la part de l'utilisateur (même s'il n'y a rien de très compliqué en soi) et surtout qu'un certain temps soit passé par l'utilisateur en recherche des réglages optimaux. Seul bémol : le flou focal obtenu n'est en rien réaliste, et répond à des critères artistiques correspondant à l'idée subjective que l'utilisateur se fait d'un bon effet de flou focal.

1.2 Méthode 3D : le raytracing

Dans cette méthode, nous en reviendrons à la définition du raytracing pour comprendre comment le flou focal est simulé. En temps normal, pour chaque pixel de l'image rendue, un rayon est lancé depuis la caméra, et, s'il rencontre un objet (disons de couleur rouge), le moteur de rendu affiche un point rouge sur l'image finale. Lorsque l'on cherche à simuler un effet de flou focal en raytracing, on commence par définir l'ouverture de la caméra ; plus celle-ci sera faible, moins il y aura de flou. Au contraire, plus l'ouverture sera élevée, plus l'image pourra être floue. Le principe va donc consister à définir, pour chaque pixel, un cône (dont l'origine coïncide avec celle de la caméra) et dont l'angle d'ouverture est lié à la valeur d'ouverture spécifiée à la caméra. Au lieu de lancer un rayon unique depuis la caméra jusqu'au pixel visé, le moteur de rendu lancera un certain nombre de rayons, aléatoirement, mais tous situés à l'intérieur du cône. La couleur de chacun des rayons sera moyennée et le résultat affiché sur l'image finale. Il en résulte une propriété intéressante : à valeur d'ouverture égale, plus vous lancerez de rayons, plus vous obtiendrez un flou progressif. Si vous voyez apparaître un phénomène de grain, c'est certainement dû au fait que le moteur de rendu ne lance pas un nombre suffisant de rayons.

Figure 07 : Exemple d'un rendu de flou focal par Raytracing avec Yafray (rendu effectué avec option d'illumination globale par la méthode de Monte-Carlo)



L'avantage de cette méthode est qu'elle est extrêmement simple à mettre en œuvre : vous devez définir la distance focale, l'ouverture de la caméra, et, enfin, décider du nombre de rayons à lancer pour obtenir un résultat le moins « granuleux » possible. En revanche, l'inconvénient principal est de taille : pour définir la couleur d'un pixel, vous lancez un nombre considérable de rayons ; et, au plus, vous souhaiterez atténuer le phénomène de granularité affectant vos images, au plus vous augmenterez le nombre de rayons et, par conséquent, les temps de rendu. Si ce compromis peut être satisfaisant dans le cadre du rendu d'une image fixe, il devient un certain handicap dans le cadre du rendu d'une animation.

2. Le flou focal avec Yafray

Le paramétrage de l'effet de flou focal va se réaliser dans le bloc `<camera>` du fichier `.xml` de votre scène, tandis que la qualité de l'effet de flou sera déterminée par les paramètres d'anti-crénelage (*anti-aliasing*) relatifs au bloc `<render>`. Les codes que nous allons tester sont les suivants, mais seules les parties mises en évidence vont faire l'objet de nos expérimentations.

```
<camera name="MAINCAM" type="perspective" resx="400" resy="300"
  focal="1.093750" aspect_ratio="1.000000"
  dof_distance="7.500000" aperture="0.200000" use_qmc="on"
  bokeh_type="disk1" bokeh_bias="uniform" bokeh_rotation="0.000000" >
            <from x="7.481132" y="-6.507640" z="5.343665" />
            <to x="6.826270" y="-5.896974" z="4.898420" />
            <up x="7.163761" y="-6.195171" z="6.239008" />
</camera>

<render camera_name="MAINCAM"
            raydepth="5" gamma="1.000000" exposure="0.000000"
            AA_passes="8" AA_minsamples="64"
            AA_pixelwidth="1.500000" AA_threshold="0.050000"
            background_name="world_background" >
            <outfile value="/tmp/YBtest.tga" />
</render>
```

2.1 dof_distance

Ce paramètre permet de définir le point focal, c'est-à-dire le point virtuel de l'espace sur lequel la caméra est concentrée. Ce point sera affiché avec une netteté totale. Les surfaces situées en amont ou en aval du point focal sur la ligne de vue de la caméra deviennent progressivement plus floues à mesure qu'elles s'éloignent du point focal. Par exemple, sur la figure 08, avec une valeur `dof_distance` égale à `8.5` (image de gauche), la boule rouge est pratiquement nette, car très proche du point focal, et, à mesure que l'on s'éloigne vers l'arrière-plan, les autres objets qui constituent le décor sont de plus en plus flous.

Figure 08 : dof_distance = 8.5 (image de gauche) et dof_distance = 12.5 (image de droite) ; l'influence de ce paramètre sur le flou focal est sans équivoque.

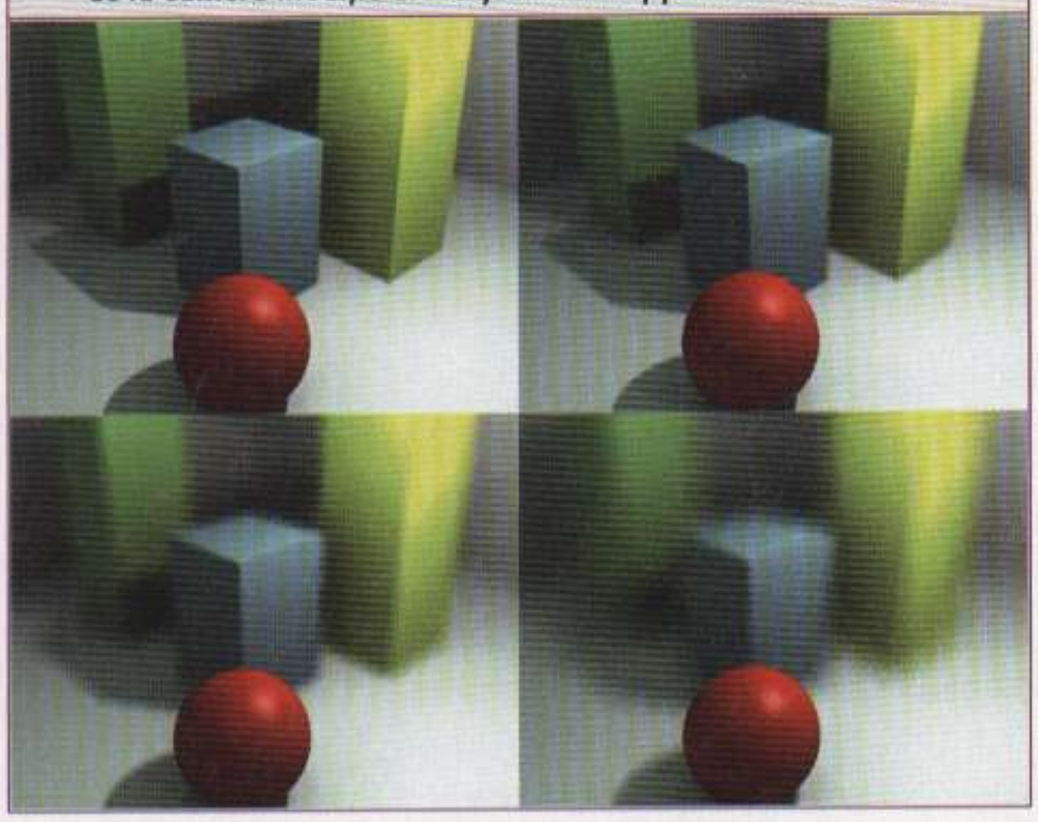
Sur l'image de droite, la valeur `dof_distance` a été passée à `12.5`, ce qui veut dire que le point focal a été éloigné vers le fond de la scène. En conséquence, les parallélépipèdes vert et jaune sont pratiquement nets, tandis que celui bleu ciel est légèrement flou. La boule rouge, au premier plan et la plus éloignée du point focal, est le volume le plus flou observé.

2.2 Aperture

Il s'agit là de l'ouverture de la lentille virtuelle de la caméra. Des valeurs élevées vont générer un flou

important, et une valeur nulle, l'absence totale de profondeur de champ. Sur les images qui suivent, les valeurs d'`aperture` sont respectivement égales à `0.1`, `0.2`, `0.5` et `1.0`. L'évolution du champ de profondeur est sans appel : il s'amenuise à mesure que l'ouverture augmente.

Figure 09 : À point focal strictement identique, la valeur d'ouverture de la caméra modifie drastiquement l'apparence de la scène.



2.3 use_qmc

Ce paramètre n'a qu'un usage assez limité. En l'activant (`use_qmc="on"`), la caméra utilisera un échantillonnage de Monte-Carlo, qui fera converger les calculs un petit peu plus vite, mais un grand nombre d'échantillons demeurera nécessaire pour avoir un flou de qualité correcte, surtout en cas d'usage des bokehs. Dans son état actuel, son intérêt est très limité, mais son usage n'induit pas de contrepartie. Vous pouvez donc l'activer à chaque fois que vous souhaiterez faire un rendu avec du flou focal. Si vous ne l'utilisez pas, un échantillonnage purement aléatoire sera utilisé lors de la phase de lancer de rayons.

3. Les bokehs

Il est assez difficile de trouver une « bonne » définition du bokeh. Disons qu'il s'agit d'un mot japonais et qu'il définit, aujourd'hui, chez nos amis photographes, l'aspect des zones floues à l'arrière-plan de leurs sujets. Le choix de la forme d'un bokeh ne devant répondre primairement qu'à des considérations esthétiques, le mieux est de tenter de les illustrer plutôt que d'en parler trop longuement. Yafray propose différents types de bokehs : `disk1`, `disk2`, `triangle`, `square`, `pentagon`, `hexagon`, `ring`. Il est assez difficile de faire la différence à l'œil nu, mais disons que le bokeh définit la « forme » d'un point hors du champ de profondeur de la caméra. Pour définir une forme de bokeh particulière, il faut recourir à l'argument suivant :

```
bokeh_type="[forme]"
```

où vous remplacerez `[forme]` par le bokeh de votre choix, par exemple `hexagon`.

La plupart (tous à l'exception de `Disk1` et `Ring`) admettent des paramètres supplémentaires, qui permettent d'ajuster la forme du bokeh, grâce au paramètre `bokeh_bias="[type]"`, où `[type]` peut être `uniform`, `center` ou `edge`.

À l'exception de `disk1` et `ring`, tous les autres bokehs peuvent se voir donné une orientation particulière grâce à l'argument `bokeh_rotation="[valeur]"`, où `[valeur]` est tout simplement une valeur angulaire.

J'ai renoncé à préparer des exemples de rendu mettant en œuvre ces différents types de bokehs, car il est difficile de les discerner à l'œil nu, et cette difficulté serait amplifiée par l'impression sur papier. Sur un même rendu, en utilisant un bokeh de type `disk1` et un autre de type `ring`, il est possible de noter une différence significative, mais c'est bien plus délicat avec les autres bokehs.

Figure 10 : Exemple d'usage d'un bokeh de type disk1 et un autre de type ring

4. L'anti-crénelage

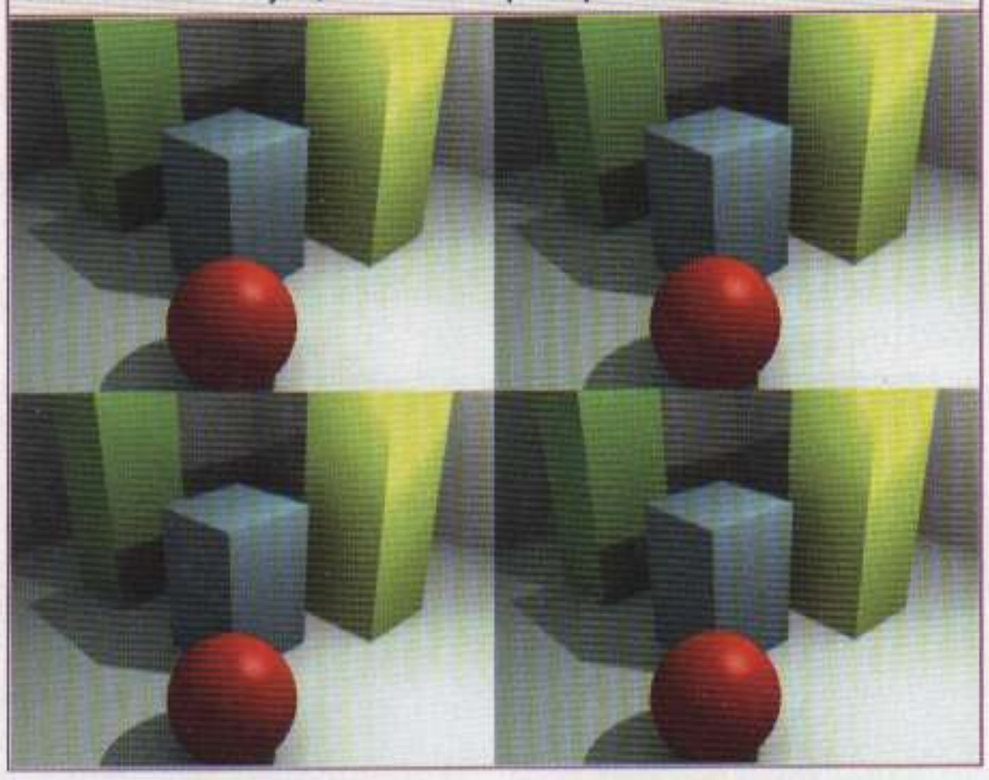
En lançant un rayon unique par pixel de votre image, plus la résolution de celle-ci sera faible, plus vous obtiendrez des formes aux contours hachés, trop nets et par conséquent fort peu réalistes. La technique d'anti-crénelage consiste à lancer, pour un même pixel, un plus grand nombre de rayons avec de très faibles déviations d'angle, afin « d'échantillonner » les pixels voisins et afficher, pour le pixel courant, la couleur qui le « fond » le mieux dans son environnement. Il en résulte alors des bords plus lisses, plus réalistes, à mesure que le taux d'échantillonnage augmente.

Figure 11 : Exemples de rendus sans anti-crénelage (image de gauche) et avec (image de droite) : les bords des objets de l'image de gauche sont précis au pixel près, et sont hachés ; sur l'image de droite, chaque pixel rendu tient compte des pixels voisins et s'harmonise plus facilement.

Yafray établit son échantillonnage en plusieurs passes, et pour chaque passe, utilise un certain nombre d'échantillons. Comme nous venons de le voir, plus le nombre d'échantillons sera élevé, plus le résultat sera lisse à l'œil nu, et cela, même avec une seule passe. Nous pouvons le voir sur la figure suivante,

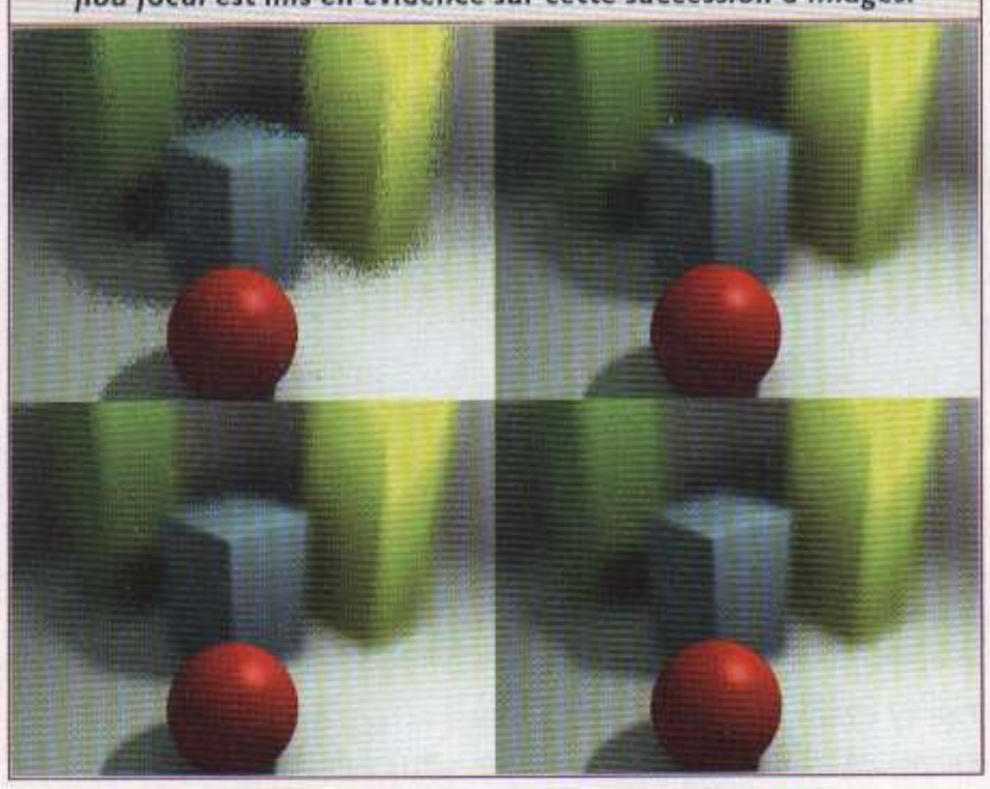
qui présente des rendus avec respectivement 0 échantillons, 2 échantillons, 8 échantillons et 32 échantillons. On voit facilement que les résultats deviennent acceptables à partir d'environ 8 échantillons, et que le résultat semble parfait aux alentours de 32 échantillons. Il n'est donc pas utile, pour un rendu classique, d'augmenter le nombre d'échantillons au-delà d'un certain niveau.

Figure 12 : Le nombre d'échantillons joue sur la fidélité de rendu des silhouettes des objets, surtout lorsqu'ils présentent des arêtes vives.



En revanche, le nombre d'échantillon par passe revêt une importance particulière lors du rendu d'une image avec un flou focal, car c'est lui qui va, principalement, atténuer la granularité de l'image. Rappelez-vous : pour déterminer la couleur d'un pixel flou, le moteur de rendu va lancer un certain nombre de rayons à l'intérieur d'un cône pointant depuis la caméra vers un pixel de l'image. Plus vous augmentez l'échantillon, plus le nombre de rayons lancés lors de cette phase sera important, et plus le pixel prendra une couleur « neutre » ou « floue » par rapport à son environnement.

Figure 13 : L'influence du nombre d'échantillons sur la qualité du flou focal est mis en évidence sur cette succession d'images.



Maintenant, en considérant que Yafray peut répéter l'opération d'anti-crénelage plusieurs fois (chaque opération est alors considérée comme une passe distincte) et d'en moyenner les résultats, on en déduit qu'augmenter le nombre de passes va également tendre à améliorer le lissage des zones floues, comme le montre l'illustration suivante. Chacune des deux images a été réalisée avec 32 échantillons par passe, mais l'image de gauche n'utilise qu'une seule passe, tandis que celle de droite en utilise quatre.

Figure 14 : Le nombre de passes d'anti-crénelage joue également son rôle dans la qualité du flou mis en place.

Toute la subtilité de ces réglages consiste donc à trouver le compromis entre le nombre de passes et le nombre d'échantillons par passe pour préserver des temps de calculs raisonnables tout en atteignant des résultats visuellement attractifs. Bien sûr, tout dépendra du niveau de flou (et donc du paramètre aperture) dans votre scène, mais vous pouvez considérer qu'avec un nombre de passes aux alentours de 4 à 8, avec et un nombre d'échantillons par passe variant de 16 à 64, vous devriez obtenir des résultats satisfaisants.

Dans le fichier `.xml` de Yafray, vous spécifierez le nombre de passes `[valeur1]` à réaliser grâce au paramètre `AA_passes="[valeur1]"`. Pour sa part, le nombre d'échantillons par passe `[valeur2]` sera défini par le paramètre `AA_minsamples="[valeur2]"`.

Conclusion

Nous en arrivons pratiquement à la fin de cette mini-série consacrée à Yafray, et nous commençons maintenant à maîtriser les différentes spécificités de ce moteur qui lui permettent de prétendre à figurer dans la catégorie des moteurs de rendu photoréalistes. Si le flou focal est un phénomène très commun dans notre monde où l'industrie visuelle est très porteuse, nous venons de voir qu'il n'était pas si difficile d'en comprendre les mécanismes pour les reproduire dans notre univers virtuel fait en images de synthèse.

Olivier Saraja,

olivier.saraja@linuxgraphic.org

LIENS

- Le site de Yafray : **www.yafray.org** [en]
- Les forums consacrés à Yafray : **www.yafray.org/forum/index.php** [en]
- La documentation de Yafray : **wiki.yafray.org/bin/view.pl/UserDoc/WebHome** [en]
- La foire aux questions (FAQ) de Yafray : **wiki.yafray.org/bin/view.pl/UserDoc/FaqEng** [en]
- De l'aide en français ? Les forums de Linuxgraphic : **www.linuxgraphic.org/forums/** [fr]

Romain Bignon

Principes et structure des réseaux IRC

Beaucoup d'entre vous connaissent l'IRC et l'utilisent fréquemment pour communiquer autour de la communauté du Libre. Je me propose de faire une explication technique de l'IRC, après avoir fait un bref historique de ce dernier.

Avant-propos

Cet article ne se veut en aucun cas être un guide de l'utilisateur, mais permet d'expliquer quels sont les principes de l'IRC et son fonctionnement au niveau protocolaire et serveurs. Il est recommandé d'avoir des notions d'IRC.

1. Qu'est-ce que l'IRC ?

IRC est un programme écrit initialement en 1988 par Jarkko Oikarien, alias WiZ, inspiré de Bitnet Relay Chat, dont le but est de relier des serveurs de différentes localités par Internet afin de permettre des échanges textuels entre des gens à travers le monde.

Au début, un seul réseau existait réunissant principalement des universitaires. Chaque serveur relié était indépendant, le réseau unique IRC était anarchique et les IRC Opérateurs avaient peu de pouvoir. Chaque serveur pouvait *linker* avec d'autres, dans la mesure où ces derniers étaient reconnus par une H-line (les désignant comme hub). Des discussions étaient lancées pour essayer de restreindre les acceptations de serveurs, mais un serveur nommé « Eris » était contre et voulait linker avec n'importe qui. Ce fut la première utilisation d'une Q-line (ligne dans la configuration refusant la connexion d'un serveur) contre Eris. Il *delinka* et créa le réseau Anet (*Anarchy Network*), qui eut peu de succès et succomba très rapidement.

Par la suite, de plus en plus de serveurs furent lancés, et beaucoup étaient rejetés. Un réseau se créa parallèlement, Undernet (*Underground Network*), et IRCD, le programme développé par Jarkko Oikarien et utilisé sur IRCnet, fut cloné pour la première fois en « IRCU ». IRCD était, à l'époque, à la version 2.7 et venait à peine d'acquérir les salons '#'. Dalnet naquit également.

Cependant, en été 1996, les désaccords continuèrent dans le réseau IRCnet. Le premier était sur la question des abus suite aux *netsplits* qui n'étaient pas négligeables (*takeovers* et prises d'identité). Deux solutions furent alors envisagées : Le *Nick/Channel Delay*, qui empêchait de prendre un pseudo ou de joindre un salon supprimé X minutes avant sa suppression, et le *timestamp*, qui marquait chaque *user* et salon de son heure de création et qui, lors du link d'un serveur, si deux users avaient le même pseudo, déconnectait le plus jeune connecté, et qui, si jamais deux salons n'avaient pas le même timestamp, redonnait tous les paramètres du plus vieux, notamment en enlevant le statut d'opérateur des opérateurs du plus jeune salon.

Le second désaccord résidait dans le pouvoir des IRC Opérateurs. Les Européens cherchaient à laisser les salons s'autogérer et à donner le moins de pouvoir aux IRC Opérateurs, alors que les IRC Opérateurs des États-Unis refusaient de perdre leur pouvoir de */kill* (déconnecter un user d'un serveur).

Ceci créa une frontière Atlantique. En effet, un gros hub des USA partageait le second avis par rapport au pouvoir des IRC Opérateurs, et les petits serveurs des États-Unis, craignant de perdre leur link, prirent la même position. Le propriétaire du hub était très désavoué particulièrement en Europe à cause de son jeune âge et de problèmes antérieurs. De plus, il menaçait une rupture USA/Europe, empêchant donc toute progression n'allant pas dans son sens. C'est pourquoi les principaux serveurs de l'opposition commencèrent un projet secret de « nouveau réseau », dans le but de réunir la majeure partie des serveurs actuels et de se débarrasser du hub en question. Cependant, ils furent trahis et le projet échoua. Une nuit, alors que l'« opposition » avait une discussion sur un canal concernant le projet pour essayer de fixer une date, le possesseur du hub le sut et delinka avec l'Europe sans prévenir quiconque. Les serveurs durent prendre partie, et l'Amérique et l'Europe de l'Ouest se rallièrent au nouveau réseau du hub, qui se nomma « Efnet ». De l'autre côté, les serveurs notamment Finlandais décidèrent de garder l'appellation IRCnet, en raison des origines finlandaises d'IRC. Cet épisode notable s'appelle le « Grand Split », semblant sorti de Dallas :).

Avec le décollage d'Internet, de très nombreux réseaux virent le jour, et la liste actuelle est innombrable. Les réseaux principaux internationaux sont Quakenet, IRCnet, Undernet et EFnet.

2. Quels programmes ?

Avant de démarrer une description détaillée de l'IRC, voici la liste des programmes à utiliser :

- clients : mIRC, xchat, irssi, bitchx, Konversation (kde), Trillian, d'autres... **[1]** ;
- serveurs : IRCU, bahamut, UnrealIRCD, et plein d'autres... **[2]**

Tous ces programmes sont compatibles Linux, sauf mIRC, que j'ai tout de même cité dans cette liste, étant le client le plus utilisé et le plus complet existant.

3. Le principe

3.1 Pour les utilisateurs

Toute personne utilisant IRC saura qu'un utilisateur peut être reconnu par une chaîne sous forme **pseudo!ident@hostname** :

- Le pseudo choisi par l'utilisateur.
- L'*ident* donné en réponse de son *identd* (port 113), et sans réponse de celui-ci, un mot configurable dans le client.
- L'*hostname* de l'utilisateur. Suivant le programme serveur utilisé par le réseau, celui ci peut être partiellement crypté.

Le protocole est basé sur des espaces de discussion appelés des « canaux » (autres appellations : *channels*/salons/*chans*). Toute personne peut créer un canal en le rejoignant et bénéficie d'un statut « opérateur » qui lui donne des privilèges sur les autres utilisateurs rejoignant son canal, permettant entre autres de nommer d'autres opérateurs, éjecter (*kicker*), voire bannir (sur un *mask* sous forme **pseudo!ident@hostname**), inviter des utilisateurs, et mettre des « modes » dont voici une liste non exhaustive :

- `t` : donne l'autorisation uniquement aux opérateurs de changer le sujet (le *topic*).
- `m` : donne l'autorisation uniquement aux opérateurs et aux *voices* (statut particulier d'utilisateur sur le canal) de parler.
- `i` : on ne peut rejoindre un salon que si l'on est invité par un opérateur.
- `n` : on ne peut parler qu'en étant dans ce salon.
- `k <clef>` : on ne peut rejoindre le salon qu'en connaissant la clef.
- `l <limite>` : limite le nombre d'utilisateur dans le canal à la limite instituée.
- `s` : le salon est secret et n'apparaît pas dans la liste des salons ou dans le profil d'un utilisateur.
- `p` : le salon est privé et a, au final, la même fonction que le mode '`s`'.

Ceci sont les modes que l'on trouve en commun sur tous les programmes serveurs (nommés couramment « IRCD », de *IRC daemons*), sachant que suivant le programme, il peut y en avoir d'autres ou pas. Notez en particulier sur certains le statut *halfop* (semi-opérateur) qui ne donne que le pouvoir d'éjecter et de bannir à celui qui le possède, statut que l'on retrouve uniquement sur peu d'IRCD, UnrealIRCD par exemple.

Il existe plusieurs types de salons, reconnaissables par leur préfixe :

- `#` (*global*) : les salons les plus utilisés, que tout le monde connaît, et qui correspondent à la description effectuée ci-dessus.
- `&` (*local*) : ce sont les salons locaux. Ils ne sont pas transmis aux autres serveurs, vous ne verrez donc que les utilisateurs de *votre* serveur.
- `+` (*modeless*) : ces salons ne sont plus utilisés. Ils avaient la particularité de ne pas avoir de modes. Pas d'opérateurs, de voices, de modes, de possibilité d'éjection, ces salons mettaient tout le monde à égalité, mais empêchaient la répression des abus. Notez que vous pouvez reproduire un tel salon en créant un canal `#`, puis en vous enlevant le statut d'opérateur.
- Encore une fois, ceci est une liste non exhaustive, d'autres types de salons existent. D'ailleurs, actuellement, pour la petite anecdote, sur le programme « IRCU » d'Undernet, on réfléchit à la création d'un type de salon spécial pour les modes '`A`' et '`U`', créés sur ce même programme, il y a quelques années (et non appliqués (encore ?) sur Undernet), qui permettent à un opérateur de mettre un mot de passe sur un salon qui peut être utilisé pour se redonner le statut opérateur.

Il est à noter, qu'au début, les canaux étaient représentés par des numéros, on ne pouvait en rejoindre qu'un seul en même temps, et le seul moyen de partir du salon était de « rejoindre » le canal 0. C'est le topic qui permettait de donner une identité à un canal. Par la suite, ce sont d'abord les canaux '`+`' (modeless), puis les canaux `#` qui apparurent, afin de supprimer définitivement les canaux numérotés. Le nom de « canal » vient de là.

L'utilisateur peut également discuter directement à quelqu'un en « privé » ou en « notice », ces deux biais étant fonctionnellement identiques, mais que le client interprète de manière différente : notice dans la fenêtre actuelle, message privé dans une fenêtre particulière à l'utilisateur qui l'envoie.

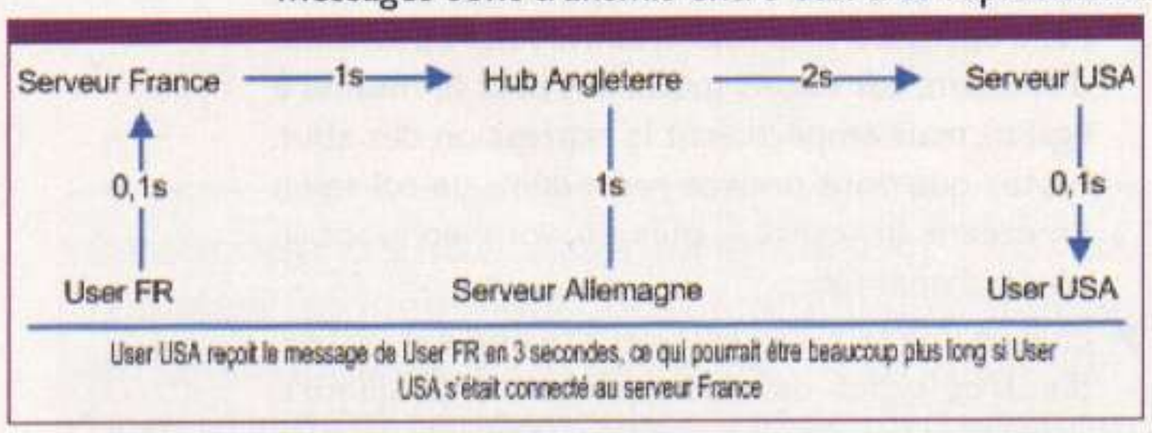
3.2 Pour les IRC Opérateurs

Chaque serveur possède des utilisateurs qui ont un statut d'« IRC Opérateurs ». Ceux-ci sont généralement désignés soit par un comité principal du réseau, soit par le propriétaire du serveur en question, suivant le mode de fonctionnement choisi par le réseau (voir chapitre sur le mode de fonctionnement). Ils ont le pouvoir de déconnecter quelqu'un du serveur, voire de le bannir de tout le réseau. Leur but en général est d'éviter toute attaque (voir chapitre sur la sécurité), la publicité massive, de s'assurer de la stabilité du serveur en général (sa connexion avec les autres serveurs), et non pas de régler les petits soucis tels que des insultes ou autres qui peuvent être évités en ignorant la personne concernée avec la commande /silence.

3.3 Pour les serveurs

L'IRC est un protocole client/serveur. Le principe est de relier plusieurs serveurs entre eux, ce qui permet à un utilisateur de se connecter à un serveur près de chez lui sans subir de *lag* (de temps de réponse excessif, supérieur à une seconde) et de pouvoir discuter avec les utilisateurs d'autres serveurs, et donc d'autres localisations, en temps réel. En effet, les serveurs étant eux sur des connexions généralement

beaucoup plus rapides que celles des clients, les messages sont transmis entre eux très rapidement.



Le but également est de permettre à chaque serveur de ne pas recevoir de surcharge d'utilisateurs. En outre, chaque message n'est transmis qu'aux serveurs nécessiteux de le connaître. Dans l'exemple, le message qui transite de User FR à User USA ne sera jamais envoyé au serveur Allemagne. Seuls les serveurs ayant un but de transmission auront connaissance de ce message.

Ces principes ont un avantage principalement pour les très gros réseaux tels que Quakenet ou Undernet qui ont des millions d'utilisateurs, et dont le but est d'éviter toute surcharge inutile. En outre, ça permet de ne pas compter sur une seule machine (en cas de plantage du programme, de la machine, perte de connexion de celle-ci, ou DDOS).

NOTE

Un crash est quelque chose de très improbable dans le sens où les IRCD sont des projets matures et programmés en général par des experts, testés sans cesse avant la mise en production.

4. Les Channel Services

4.1 Présentation

Tels que présentés dans le chapitre 3, les salons posent un vrai problème : une fois déconnecté, un opérateur ne peut récupérer son statut d'opérateur que par le biais d'un autre opérateur, et lorsqu'un salon n'a plus d'opérateur, il est impossible de récupérer le statut sans que tout le monde parte du salon (ce qui entraîne sa destruction), puis que quelqu'un le rejoigne.

C'est pourquoi une solution fut trouvée. L'idée est qu'un programme se connecte à un serveur, se faisant passer pour un serveur lui-même, mais qui ne reçoit pas de connexion, et fait croire aux autres serveurs qu'un user existe avec un statut particulier (le mode +k sur IRCU par exemple, que personne ne peut se paramétrer soi-même). Ceci permet à la fois au serveur de pouvoir mettre son utilisateur (son « robot ») opérateur sur tous les salons, d'avoir un accès IRC Opérateur, et d'être protégé contre toute forme de *deop/kick* de la part d'un autre opérateur.

Une fois ces pouvoirs acquis, ce robot peut donc recevoir des requêtes d'enregistrement de salons, et il suffira à la personne en question d'émettre une commande d'identification au robot pour que celui-ci le mette opérateur sur le salon en question. Ceci est le principe de base, et de nombreuses fonctionnalités plus ou moins utiles existent, comme la possibilité de passer par le robot pour toute forme d'opération sur le salon, la protection des opérateurs ayant un accès, des opérations de masse facilitées, une protection anti *mass-join*, etc.

L'avantage également de passer par un programme à part plutôt que d'inclure ça dans l'IRCD est qu'il y a une seule instance qui gère le tout, alors que dans le cas contraire chaque IRCD devrait se synchroniser, ainsi que de s'encombrer d'une base de donnée...

4.2 L'exemple de Z

À partir de ce principe de base, des services de plus en plus complets ont été créés. Sur le réseau francophone IRCube, l'équipe CoderZ, dont les membres sont David Cortier (alias Cesar) et moi-même, eut le projet de créer un service reprenant les idées des autres services existants, tels que l'unicité de robot de Gnuworld (X sur Undernet), la protection de pseudo de Epona (NickServ sur freenode), et la multiplicité des fonctionnalités de CS (IriX de entrechat).

Ce robot, dont le nom est Z, réunit de très nombreuses fonctionnalités, dont certaines plus ou moins inédites : commandes pour que toutes vos actions sur un salon passent par le robot, possibilité de taper directement la commande sur le salon précédée du caractère de reconnaissance '!', la protection des opérateurs ayant un accès, des opérations de masse facilitées, une protection anti mass-join, envoi de mémos sauvegardés, système de votes organisés par un administrateur du robot, historique des commandes accessible par les administrateurs, etc.

Ce sont ces services de plus en plus complets qui permettent une accessibilité plus importante à l'utilisateur lambda qui ne cherchera pas à comprendre tous les principes d'IRC et qui préférera avoir un outil beaucoup plus intuitif.

5. La sécurité

5.1 Les splits

Lorsqu'il y a une perte de connexion entre deux serveurs, la difficulté est de faire en sorte d'assurer la resynchronisation entre les serveurs pour que chacun puisse retrouver sa place et éviter la prise de pouvoir sur un salon par exemple. Ceci est très bien géré par les IRCD. Nous prendrons le cas d'IRCU, dont le système de synchronisation se fait par envoi successif de tous les utilisateurs, tous les salons (sur une ligne : timestamp, modes, opérateurs, voices, bans), ainsi que tous les topics.

Il en a déjà été question dans le premier paragraphe, lors de la guerre entre les timestamp et Nick/Channel Delay.

Le principe du timestamp est que chaque salon et chaque user a stocké en mémoire la date de création ou de connexion. Pour les salons, lors de la resynchronisation des serveurs, si le timestamp des deux salons à *merger* est identique, on fait confiance aux opérateurs et modes présents des deux cotés, et on synchronise les deux en remettant les OPS et modes de l'autre serveur. En revanche, si l'un des deux salons a un timestamp supérieur à l'autre, c'est que celui-ci est plus récent et ainsi a été recréé pendant le split. Dans ce cas-là, ce sont tous les opérateurs

et modes de l'ancien salon qui seront gardés et tous les opérateurs du salon le plus récent perdront leur statut. Pire encore, si jamais il y a sur le salon le plus vieux un mode d'inaccessibilité direct au salon, tel que le +i (*invite only*) ou le +k (*key requiered*), tous les utilisateurs du nouveau salon seront éjectés.

En ce qui concerne les utilisateurs, ça se base sur le même principe. Lors d'une resynchronisation des serveurs, si jamais deux utilisateurs possèdent le même pseudo, celui qui a le timestamp le plus gros, c'est-à-dire l'utilisateur qui s'est connecté le plus récemment, sera déconnecté du serveur.

5.2 Les DDOS

Le DOS (*Denial Of Service*) est un envoi de flux très important d'un point à un autre, visant la saturation de la ligne employée. Le DDOS (*Distributed Denial Of Service*) est l'application parallèle de multiples DOS contre un point (un serveur), en provenance d'un maximum d'origine, causant ainsi la saturation d'un très grand nombre de lignes différentes en direction de ce serveur. Ceci est un problème assez important sur IRC, et les manières de le résoudre ne sont pas très nombreuses ni forcément très efficaces :

- Avoir un nombre de serveurs relativement élevés, de différents fournisseurs d'accès, voire dans des pays différents, afin de limiter le DDOS que sur un nombre réduit de serveurs.
- Avoir une meilleure connexion que l'attaquant, ce qui empêchera la saturation, mais provoquera tout de même un ralentissement (moins de bande passante disponible).
- Repérer au fur et à mesure, lors d'une attaque, les hosts des machines effectuant trop de requêtes, et les bloquer par le firewall.
- Avoir des contacts avec des Fournisseurs d'Accès à Internet ou la police :).

De manière générale, il faut un maximum d'anticipation pour éviter que, lors d'un DDOS, le réseau ne soit [trop] atteint.

5.3 Les attaques de proxys

Contrairement aux DDOS, les attaques de proxys sont beaucoup plus faciles à éviter, dans le sens où ce sont des clients qui se connectent directement au réseau. Leur but n'est pas la saturation de la machine (sauf si le nombre de proxys est vraiment très élevé), mais plutôt de faire du *flood* sur les salons. Suivant la façon dont l'attaquant procède, il peut être plus ou moins facile de contrer ces attaques. Voici ce dont doit être doté le service qui s'occupera de détecter les attaques de ce type :

- Vérification des ports ouverts du client (1080, 8080, 80, 3128, ports les plus utilisés par les proxys). Notez que la vérification du port 80 peut poser problème si un utilisateur héberge un HTTPD sur sa machine.
- Détection de connexions en masse. Valable plutôt sur les petits réseaux où il n'est pas habituel d'y avoir beaucoup de connexions d'un coup. Il vaut mieux cependant afficher un message d'avertissement aux administrateurs plutôt que de prendre les mesures automatiquement.
- Il existe un système intitulé « BOPM », qui permet de partager une *blacklist* entre différents réseaux. Malheureusement, la liste n'est plus maintenue et ce programme est obsolète. Cependant, il est toujours intéressant de faire des alliances avec d'autres réseaux pour faire sa propre blacklist.
- Détection sur l'ident. En effet, il est fréquent que l'ident du programme utilisé soit généré de manière aléatoire (par exemple u1564 (une lettre, quatre chiffres)). En général, quand le *regex* est bien fourni, ça devient très efficace.
- Commande pour *gline* (bannir du réseau) tous les membres d'un salon (il est fréquent que les clones se retrouvent sur un salon créé par le pirate avant d'attaquer, peut être pour permettre à celui-ci d'admirer la beau *botnet* qu'il possède :)).
- Émettre un avertissement, lorsqu'un grand nombre de personnes joignent un salon d'un coup, limite gline directement.
- « Autolimite », une protection sur le service de protection des salons, qui paramètre toutes les X minutes la limite +l à nombre de membres du salon + Y pour empêcher qu'il y ait plus de Y personnes qui joignent le salon d'un coup.

Ainsi que vous pouvez le constater, la liste des solutions est assez vaste et peut être encore agrandie.

Conclusion

En conclusion, on peut considérer qu'IRC est un protocole qui a des caractéristiques plutôt élaborées, et n'a cessé d'être amélioré durant ses 18 années d'existence (il est né en même temps que moi :p). Cependant, la limite commence à être atteinte. Il est difficile de concurrencer les messageries instantanées qui proposent le son, l'image et la vidéo, fonctionnalités qu'on ne pourra jamais retrouver sur IRC. En effet, le plus gros problème réside dans le fait que, de par la multiplication des clients IRC, il sera impossible de pouvoir faire des modifications fondamentales dans le protocole client<->serveur, au risque de rendre le serveur inutilisable par les clients actuels.

Néanmoins, IRC reste quand même très répandu, notamment chez les acteurs du Libre, pour les avantages des salons pas suffisamment développés sur les messages instantanées, et par son caractère plus sobre (passons sur les horribles couleurs que l'on doit à mIRC:p).

Romain Bignon,

Progs@coderz.info

RÉFÉRENCES

- [1] http://fr.wikipedia.org/wiki/Liste_de_clients_IRC
- [2] http://fr.wikipedia.org/wiki/Ircd

Cyril Meusnier

Supervision avec OCS Inventory NG et GLPI

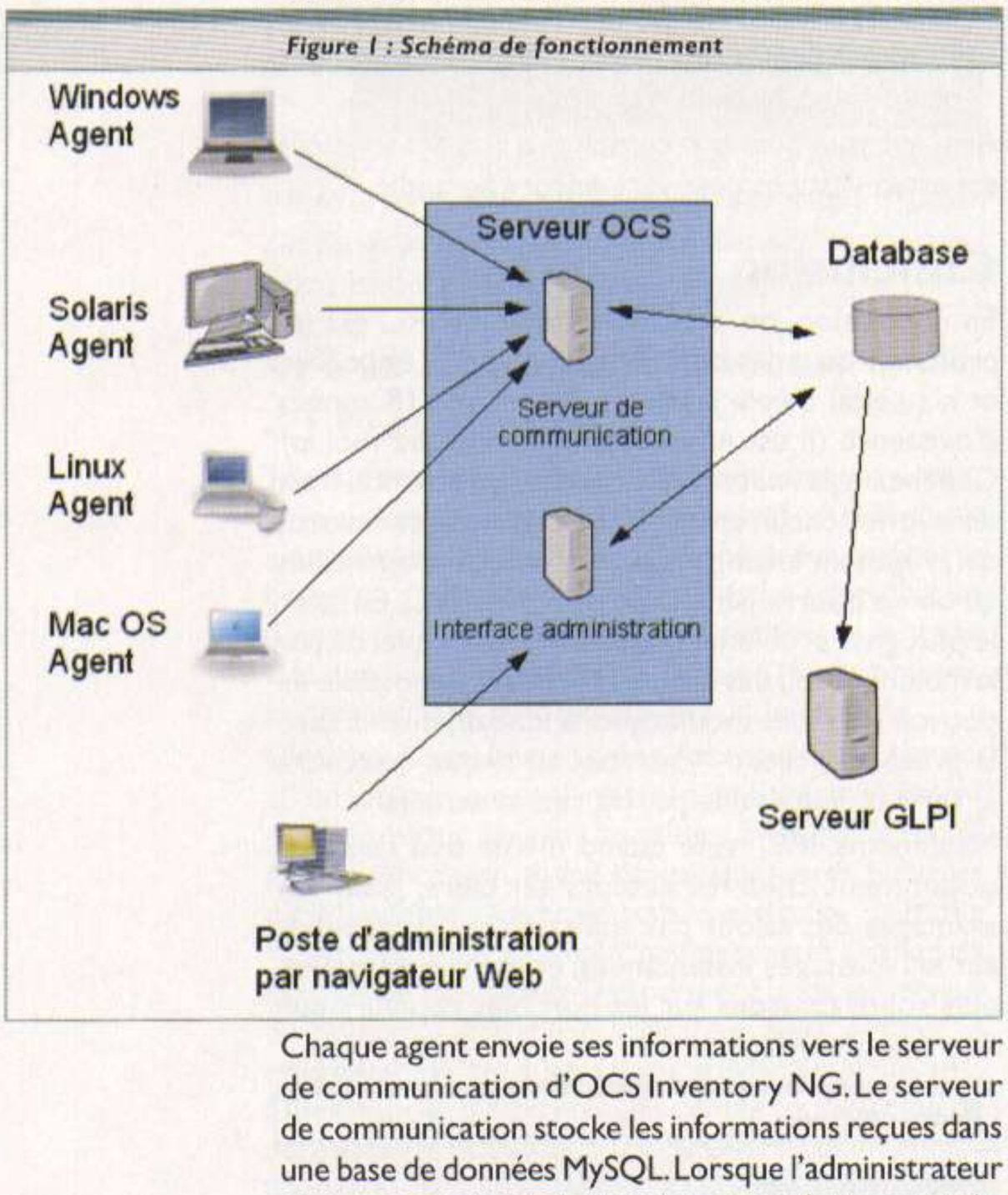
Dans une entreprise, il est important de connaître son parc informatique. L'un des plus connus dans le monde Unix est Nagios. L'inventaire reste une tâche longue et fastidieuse. C'est pour cela que des Logiciels libres comme OCS Inventory NG et GLPI existent.

I. OCS Inventory NG

I.1 Présentation

Commençons par le commencement logique du but de cet article. OCS Inventory NG se présente sous la forme d'un serveur et d'un client. Le client se présente sous la forme d'un programme à installer sous Linux, Windows, AIX, Novell ou encore MAC OS. J'ai essayé de présenter son fonctionnement dans le schéma ci-dessous.

Figure 1 : Schéma de fonctionnement



Chaque agent envoie ses informations vers le serveur de communication d'OCS Inventory NG. Le serveur de communication stocke les informations reçues dans une base de données MySQL. Lorsque l'administrateur se connecte, les informations sont, bien entendu, renvoyées pour consulter les enregistrements. GLPI, quant à lui, se contentera de rapatrier les données de la base de données dans sa table. On retrouvera dans la table des informations comme : BIOS, carte mère, RAM, logiciels, système d'exploitation, lecteur CD/DVD, graveur CD/DVD, moniteur, carte son, carte vidéo, imprimante connectée...

I.2 Installation

Avant d'installer, il y a une préparation simple du serveur à faire, à savoir installer Apache2, PHP4, MySQL. Pour ce faire, il suffit d'exécuter les commandes suivantes :

```
apt-get install apache2 php4 mysql-server
apt-get install php4-mysql
apt-get install libapache2-mod-php4
```

puis d'activer le support PHP pour Apache2 et de relancer le serveur Web :

```
a2enmod php4
/etc./init.d/apache2 restart
```

OCS Inventory NG se présente sous forme d'un fichier compressé. Pour l'installer, il suffit de le décompresser dans un répertoire de la machine. Pour raison pratique, il est conseillé de ne pas le décompresser directement dans le répertoire `/var/www/`, mais plutôt dans un répertoire du style `/usr/local/src/`. Dans le premier, vous ferez un lien vers le deuxième. Mais vous pouvez également le décompresser directement dans le répertoire d'Apache `/var/www`.

```
cd /usr/local/src/
wget http://belnet.dl.sourceforge.net/sourceforge/ocsinventory/
OCSNG_LINUX_SERVER_1.0RC3-1.tar.gz
tar -zxvf OCSNG_LINUX_SERVER_1.0RC3-1.tar.gz
cd /var/www
ln -s /usr/local/src/OCSNG_LINUX_SERVER_1.0RC3-1/ocsreports/  ocs
cd /usr/local/src/ OCSNG_LINUX_SERVER_1.0RC3-1/
sh setup.sh
```

On va donc se placer dans le répertoire où l'on souhaite télécharger le paquet. Ensuite, nous récupérons le paquet directement sur le site avant de le décompresser. Nous créons ensuite le lien dans le répertoire d'Apache2. Nous entrons dans le répertoire et lançons l'installation. Nous avons laissé les options par défaut de l'installation. Changez, bien évidemment, si vous pensez qu'il faut changer. Les images suivantes montrent une partie des fenêtres durant l'installation.

```
+------------------------------------------------------+
| Checking for Apache user account...                  |
+------------------------------------------------------+

Which user account is running Apache web server (www-data) ?
OK, Apache is running under user account www-data :-)


+------------------------------------------------------+
| Checking for Apache group...                         |
+------------------------------------------------------+

Which user group is running Apache web server (www-data) ?
OK, Apache is running under users group www-data :-)


+------------------------------------------------------+
| Checking for PERL interpreter...                     |
+------------------------------------------------------+

OK, PERL Intrepreter found at (/usr/bin/perl) :-)


Do you wish to setup Communication server on this computer (y)/n)?_
```

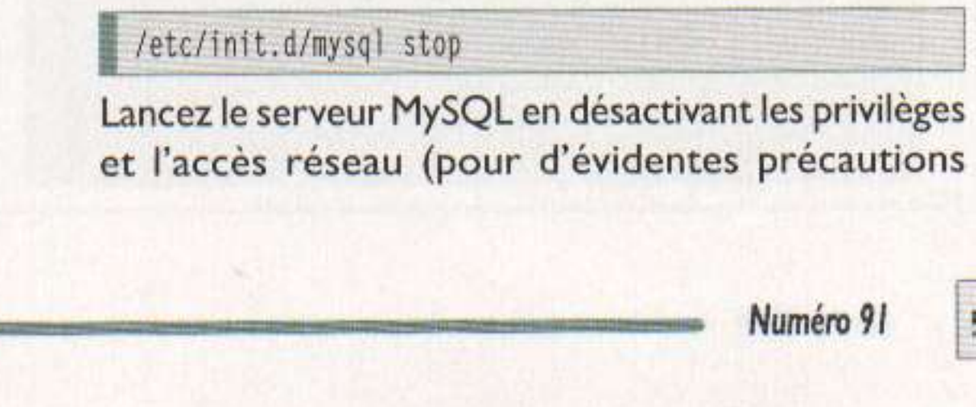
```
Setup has found Apache Include configuration directory is
//etc/apache2/conf.d/.
If you are not using Include directive, please enter 'no'.
Where is Apache Include configuration directory [//etc/apache2/conf.d/] ?no
Not using Apache Include configuration directory.
Configuration will be written to Apache main configuration file
//etc/apache2/apache2.conf.

+--------------------------------------------------------+
| Checking for Apache mod_perl version...                |
+--------------------------------------------------------+

Checking for Apache mod_perl version 1.99_22 or higher
Checking for Apache mod_perl version 1.99_21 or previous
Setup is unable to determine your Apache mod_perl version.
Apache must have module mod_perl enabled. As configuration differs from
mod_perl 1.99_21 or previous AND mod_perl 1.99_22 or higher, Setup must
know which release Apache is using.
You can find which release you are using by running the following command
   - On RPM enabled OS, rpm -q mod_perl
   - On DPKG enabled OS, dpkg -l libapache-mod-perl
Enter 1 for mod_perl 1.99_21 or previous.
Enter 2 for mod_perl 1.99_22 and higher.
Which version of Apache mod_perl the computer is running ([1]/2) ?_
```

```
Fixing directories and files permissions.

Configuring IPDISCOVER-UTIL Perl script.

Installing IPDISCOVER-UTIL Perl script.

Fixing permissions on IPDISCOVER-UTIL Perl script.

+--------------------------------------------------------+
| OK, Administration server installation finished :-)    |
|                                                        |
| Point your browser to http://server/ocsreports to      |
| configure database server and create/update schema.    |
+--------------------------------------------------------+


Setup has created a log file setup.log. Please, save this file.
If you encounter error while running OCS Inventory NG Management server,
we can ask you to show us his content !

DON'T FORGET TO RESTART APACHE DAEMON !

Enjoy OCS Inventory NG :-)
```

Il se peut que, durant l'installation, vous ayez des messages disant que des modules manquent. Pour résoudre ce problème, exécuter les lignes suivantes :

```
apt-get install libapache-dbi-perl
apt-get install libdbd-mysql-perl
apt-get install libcompress-zlib-perl
apt-get install libxml-simple-perl
apt-get install libnet-ip-perl
```

Une fois l'installation terminée, rendez-vous à l'adresse : **http://adresse_vserver/ocs/**. La page suivante apparaît :



NOTE

Si comme ici vous avez un message d'avertissement concernant GD de PHP, lancez :

```
apt-get install php4-gd
```

Il faut, sur cette page, renseigner les différents champs correspondant aux identifiants que vous avez entrés lors de l'installation du serveur MySQL. Si vous avez perdu votre identifiant, pas de panique. Il existe un moyen de réinitialiser son mot de passe *root*. Tout d'abord, arrêtez le *daemon* qui tourne :

```
/etc/init.d/mysql stop
```

Lancez le serveur MySQL en désactivant les privilèges et l'accès réseau (pour d'évidentes précautions sécuritaires...). En effet, sans cette option, votre base est ouverte à l'ensemble du monde sans restriction...

```
sudo mysqld_safe --skip-grant-tables --skip-networking &
```

Connectez-vous sur le serveur MySQL qui tourne :

```
mysql mysql
```

Dans le shell MySQL, définissez votre nouveau mot de passe :

```
UPDATE user SET password=PASSWORD('votre_nouveau_mot_
de_passe') WHERE user="root" AND host="localhost";
```

Sortez du client MySQL :

```
exit
```

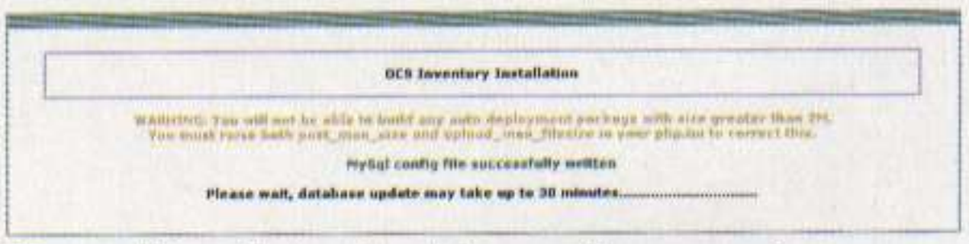
Arrêtez votre instance MySQL :

```
sudo mysqladmin shutdown
```

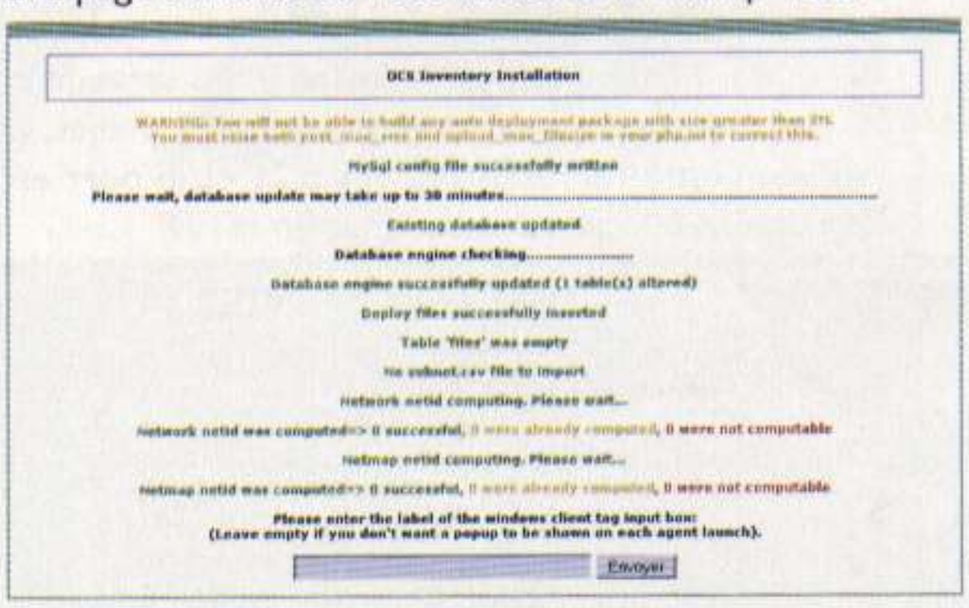
Lancez votre instance « normale » MySQL :

```
sudo /etc/init.d/mysql start
```

Complétez ensuite les paramètres de la page de configuration MySQL. Après validation, une autre fenêtre va s'afficher montrant la suite de l'installation :



Si toute l'installation s'est bien passée, vous arriverez à la page suivante où vous laisserez le champ vide.



Cette page signifie que la base de donnée a bien été mise à jour. Cliquez donc directement sur « envoyer ». Revalidez la fenêtre suivante et vous entrerez dans la page d'authentification d'OCS Inventory NG. Le *login* par défaut est : « admin » et « admin »

1.3 Installation de la partie agent

Dans cette partie, nous allons vous expliquer l'installation de l'agent d'OCS Inventory NG sur une machine Linux. Tout d'abord, il vous faut le fichier `tar.gz` de l'agent à installer que vous trouverez sur le site officiel. Une fois le fichier téléchargé, nous le placerons dans le répertoire `/usr/local/src/`. Puis, nous le décompresserons et lancerons le script `setup.sh` :

```
tar xzvf OCSNG_LINUX_AGENT_1.0RC3.tar.gz
cd OCSNG_LINUX_AGENT_1.0RC3
sh setup.sh
```

Dans la première image, le script a détecté une installation antérieure. Il demande donc si nous voulons le réinstaller. Nous répondons donc « oui » :

```
+------------------------------------------------------------+
|                                                            |
| Welcome to OCS Inventory NG Agent setup !                  |
|                                                            |
+------------------------------------------------------------+


+------------------------------------------------------------+
| Checking for previous installation...                      |
+------------------------------------------------------------+

Previous installation of OCS Inventory NG agent was found.

+------------------------------------------------------------+
| Checking for supplied parameters...                        |
+------------------------------------------------------------+

No parameter found
OCS Inventory NG Agent setup running in user interactive mode

Previous installation of OCS Inventory NG agent was found.
This installation was using OCS Inventory NG Communication Server on host
192.168.0.21 and port 80.
Do you wish to re-install/upgrade existing installation ([y]/n) ?
```

```
| Checking for supplied parameters...                        |
+------------------------------------------------------------+

No parameter found
OCS Inventory NG Agent setup running in user interactive mode

Previous installation of OCS Inventory NG agent was found.
This installation was using OCS Inventory NG Communication Server on host
192.168.0.21 and port 80.
Do you wish to re-install/upgrade existing installation ([y]/n) ?

+------------------------------------------------------------+
| Checking for OCS Inventory NG Agent running method...      |
+------------------------------------------------------------+

OCS Inventory NG Agent can be run through 2 methods:
- local: inventory will be generated locally to a file, without
         interacting with Communication Server. Inventory results
         must then be imported manually into the server through
         Administration Console.
- http: Agent can connect to Communication Server and will interact
        with it to know what is has to do (inventory, ipdiscover,
        deployment...)
Which method will you use to generate the inventory ([http]/local) ?
```

Nous laisserons les valeurs par défaut. A la question suivante, il faudra entrer l'adresse IP du serveur et le port de communication. Dans notre projet, le serveur est à l'adresse 192.168.0.21 et le port est le numéro 80, car nous sommes en HTTP.

```
OK, OCS Inventory NG Agent configuration files setup successfully ;-)


+------------------------------------------------------------+
| Installing OCS Inventory NG Agent cron configuration...    |
+------------------------------------------------------------+

OCS Inventory NG Agent cron configuration file already exist, skipping


+------------------------------------------------------------+
| Lauching OCS Inventory NG Agent...                         |
+------------------------------------------------------------+

OK, OCS Inventory NG Agent runs successfully ;-)


Setup has created a log file setup.log. Please, save this file.
If you encounter error while running OCS Inventory NG Agent,
we can ask you to show us his content !

Enjoy OCS Inventory NG ;-)
debgui:/usr/local/src/OCSNG_LINUX_AGENT_1.0RC3# OCSNG_LINUX_AGENT_1.0RC3
```

Dans le fichier README, vous avez la liste des dépendances à installer. Si vous avez des problèmes lors de l'installation, c'est que les dépendances ne sont pas installées. Ensuite, l'installation se déroulera et OCS Inventory sera opérationnel. La commande ocsinv permet d'envoyer les données au serveur :

```
debgui:/usr/local/src/OCSNG_LINUX_AGENT_1.0RC3# ocsinv
Fri Dec 22 22:30:54 2006 => Transmission...done.
Fri Dec 22 22:30:54 2006 => Account infos up to date
Fri Dec 22 22:30:54 2006 => Terminated... :-)
Fri Dec 22 22:30:54 2006 => Execution time : 16 secs
debgui:/usr/local/src/OCSNG_LINUX_AGENT_1.0RC3#
```

2. GLPI

2.1 Présentation

GLPI est une application libre, distribuée sous licence GPL, destinée à la gestion de parc informatique et de *helpdesk*. GLPI est composé d'un ensemble de services web écrits en PHP qui permettent de recenser et de gérer l'intégralité des composantes matérielles ou logicielles d'un parc informatique, et ainsi d'optimiser le travail des techniciens grâce à une maintenance plus cohérente.

2.2 Installation

Après avoir installé ce qu'il nous fallait au préalable, c'est-à-dire la partie serveur d'OCS, il nous faut dorénavant installer GLPI (Gestion Libre de Parc Informatique). L'installation s'effectue comme pour la partie serveur d'OCS Inventory. Il suffit de télécharger le fichier :

```
cd /usr/local/src/
wget http://www.glpi-project.org/IMG/gz/glpi-0.68.2.tar.gz
```

puis de la décompresser :

```
tar xzvf glpi-0.68.2.tar.gz
```

et de faire un lien symbolique sur le répertoire dans la partie Apache :

```
cd /var/www
ln -s /usr/local/src/glpi-0.68.2  glpi
```

Votre serveur est désormais installé. Il ne reste plus qu'à le configurer en allant directement sur l'URL **http://192.168.0.21/glpi/**. Vous arriverez à cette page ci :

Figure 2 : Page d'accueil avant installation. Après avoir cliqué sur « OK », vous aurez les fenêtres suivantes :

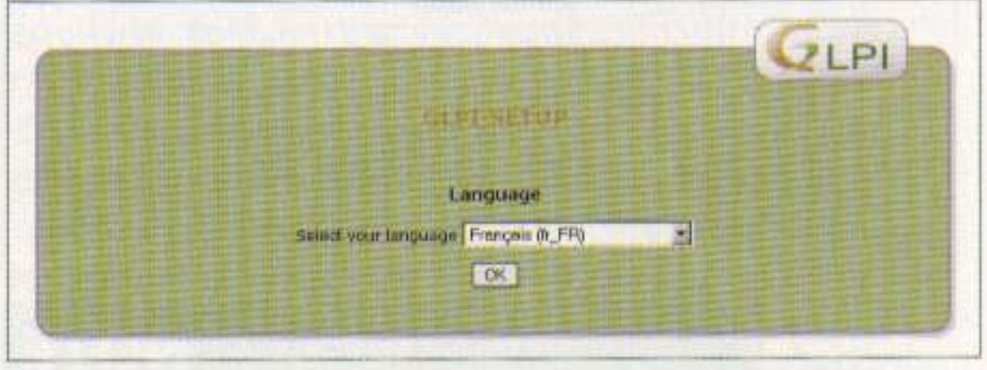


Figure 3 : Page d'acceptation

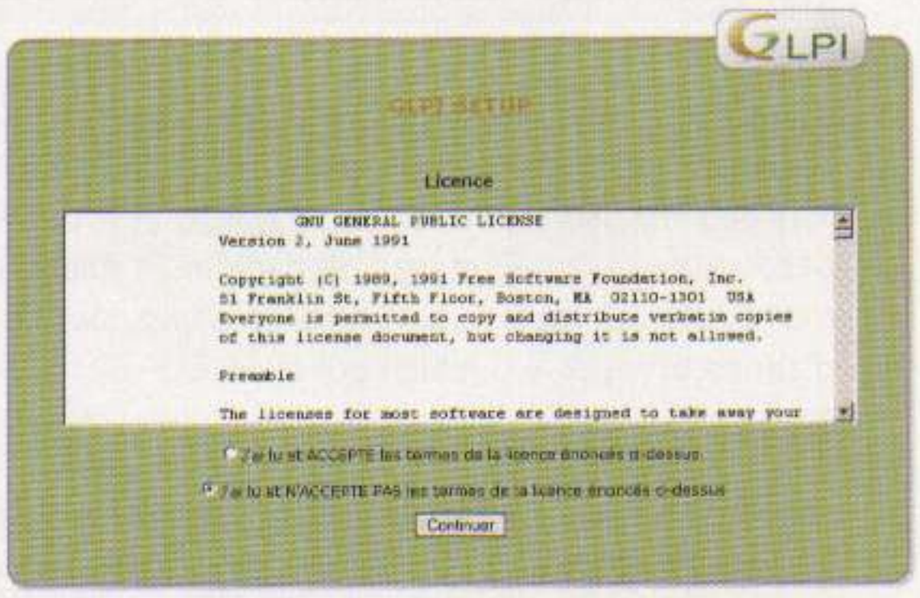


Figure 4 : Choix du type d'installation

La page qui suit montre les paramètres ou droits qui doivent avoir lieu à savoir :

```
dans php.ini : memory_limit = 16M
```

pour changer les droits, les personnes avancées préfèreront faire :

```
chown -R www-data:www-data glpi
```

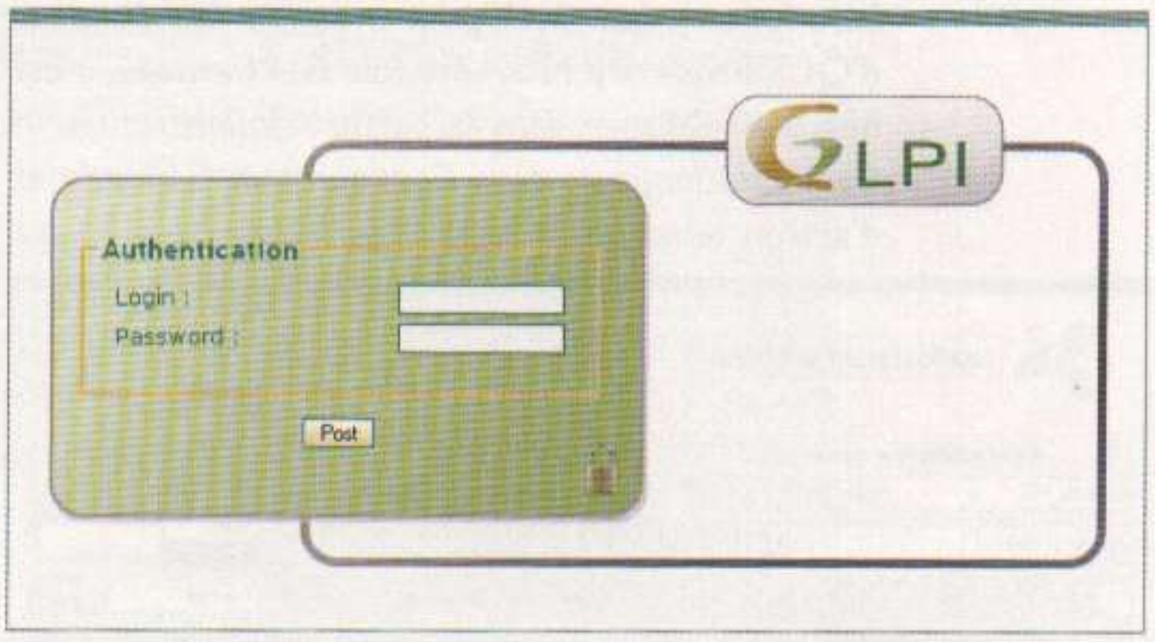
Cela est plus propre du point de vue sécurité. La suite de l'installation va créer la base de données MySQL que nous appellerons « glpi ». L'installation va se charger de générer les fichiers de configuration.

Figure 5 : Paramètres base MySQL

Figure 6 : Création table MySQL

Figure 7 : Page de fin d'installation

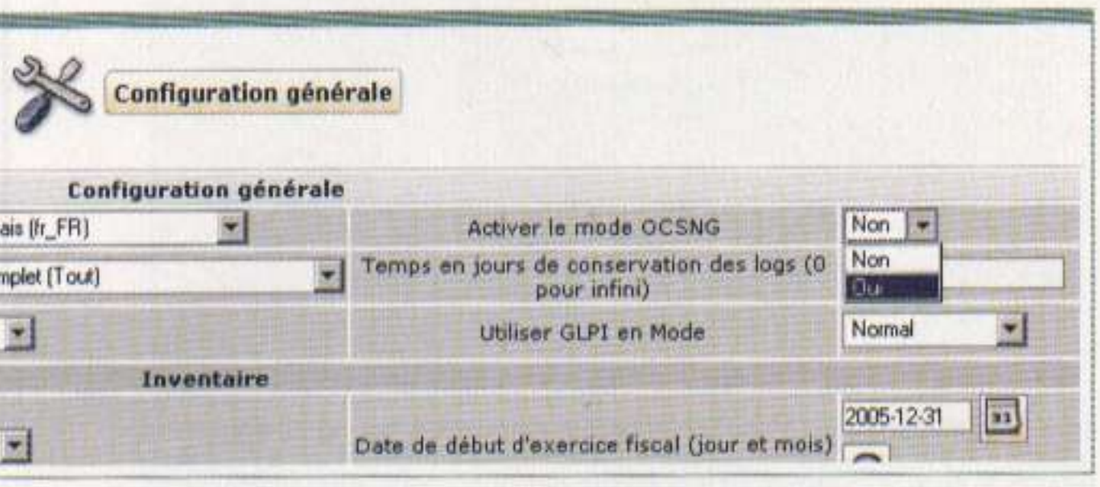
Une fois l'installation terminée, vous arriverez à la page suivante d'authentification.



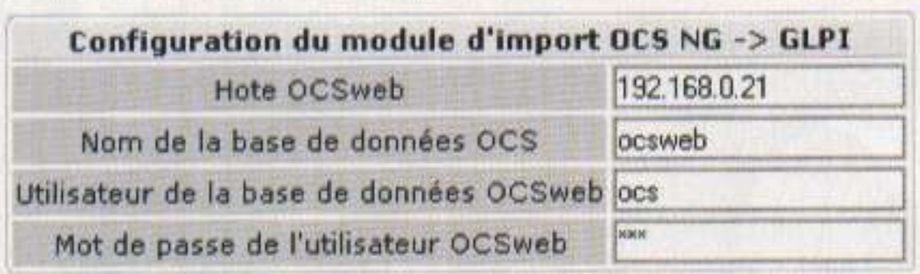
Les identifiants par défaut sont « glpi » et, en mot de passe, « glpi ». Maintenant que nous sommes connectés, il faut configurer quelques paramètres afin que GLPI puisse interagir avec OCS Inventory NG.

2.3 Configuration

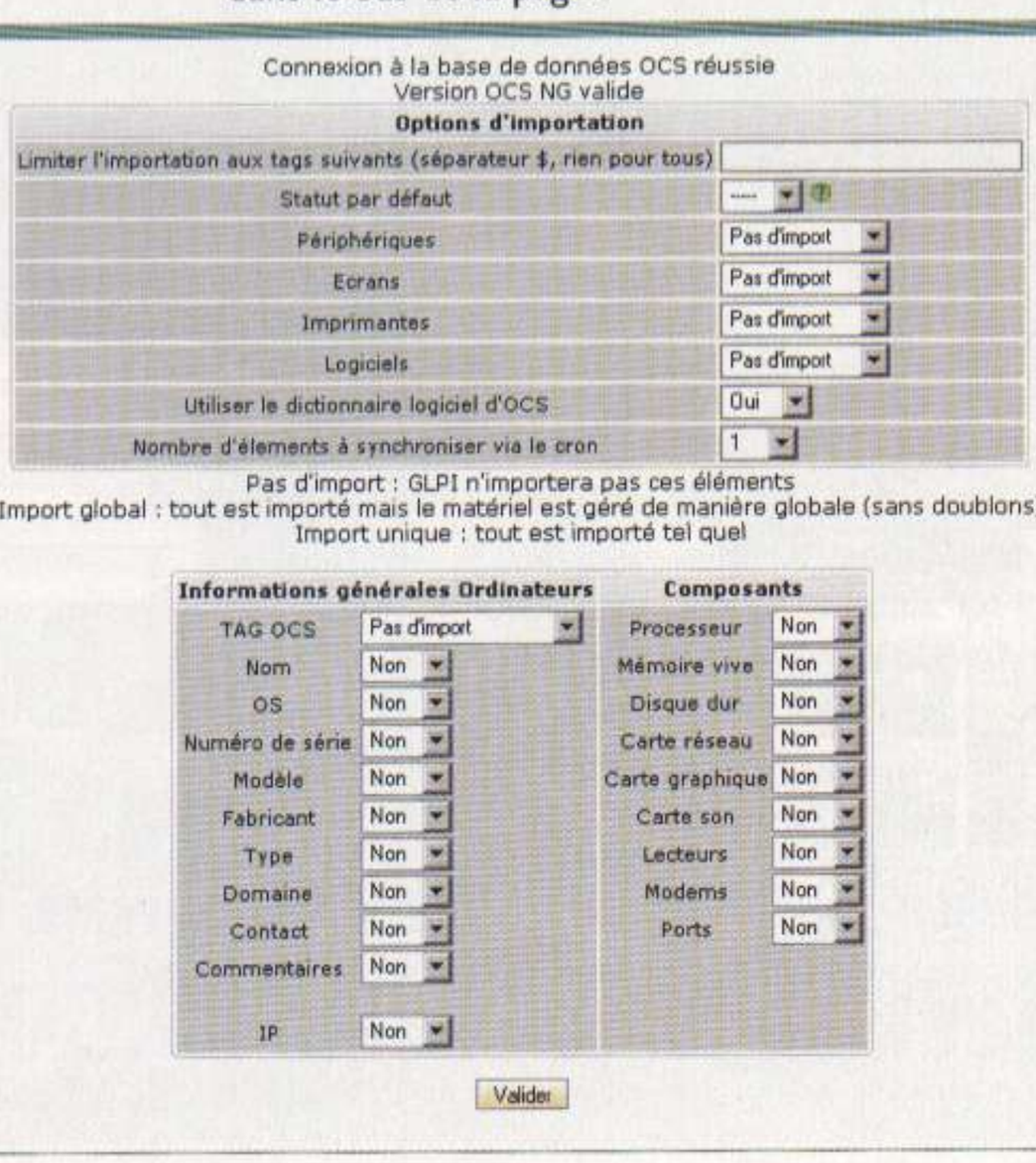
Dans cette partie, nous allons vous montrer comment faire fonctionner GLPI pour importer les données d'OCS Inventory NG. Une fois GLPI installé, il est nécessaire d'aller dans la partie *Administration -> Configuration*, puis dans *Configuration Générale* et d'activer le mode OCS.



Une fois la page validée, il vous faut entrer les paramètres pour accéder à la base de données d'OCS.

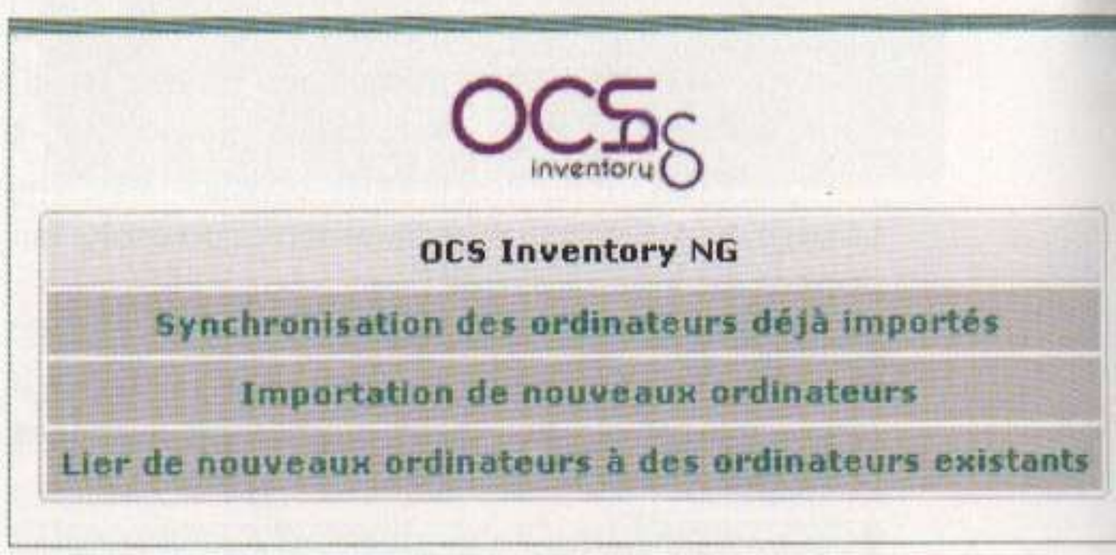


Si votre serveur est sur une machine autre, il vous suffit de rentrer l'adresse IP. Sinon, si, dans notre cas, les deux serveurs sont dans le même *vserver*, remplacez « 192.168.0.21 » par « localhost ». Le bouton « valider » fera apparaître une autre partie dans le bas de la page.

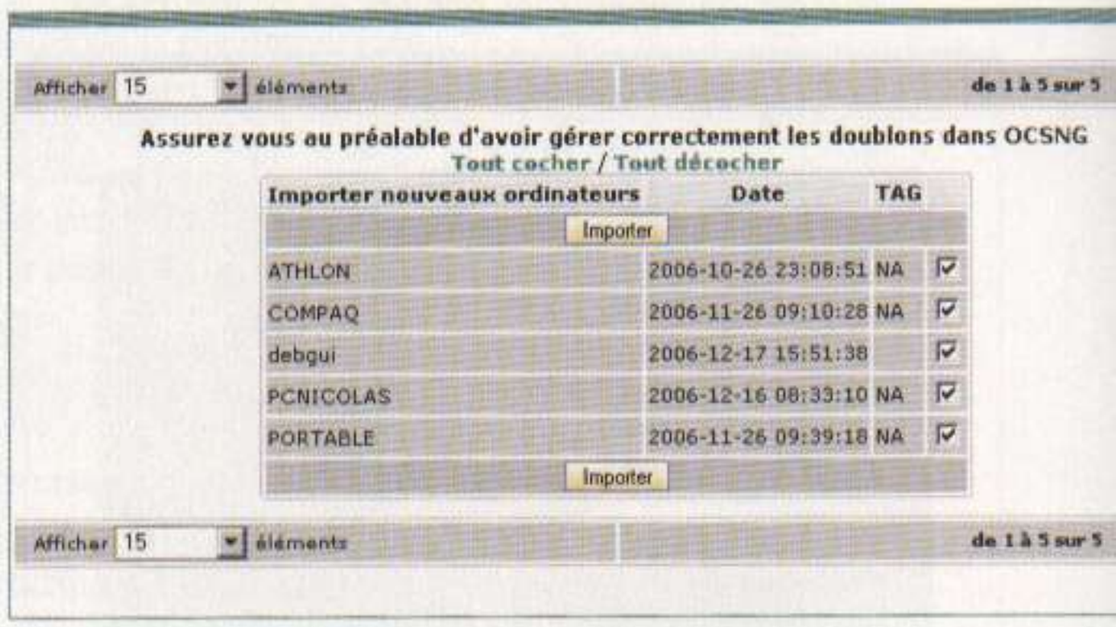


2.4 Utilisation

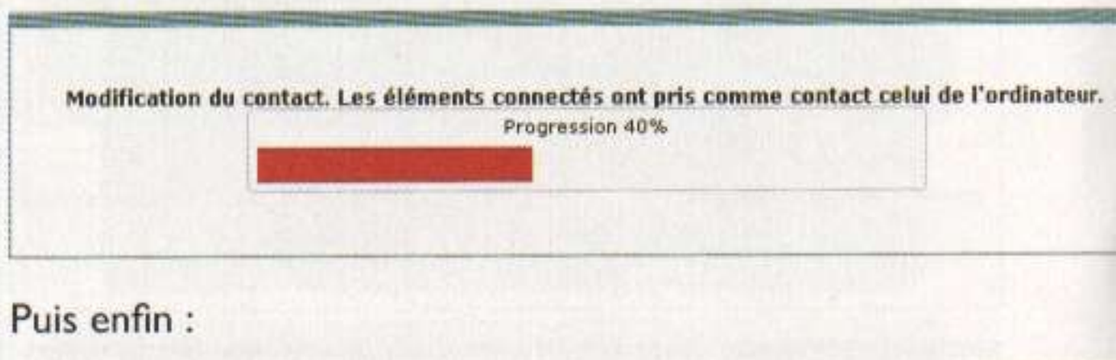
Dans cette partie, vous pouvez définir ce que vous désirez rapatrier d'OCS. La fin de cela fera apparaître, dans *Outils*, un menu *OCSNG*. C'est là que la partie OCS se trouve.



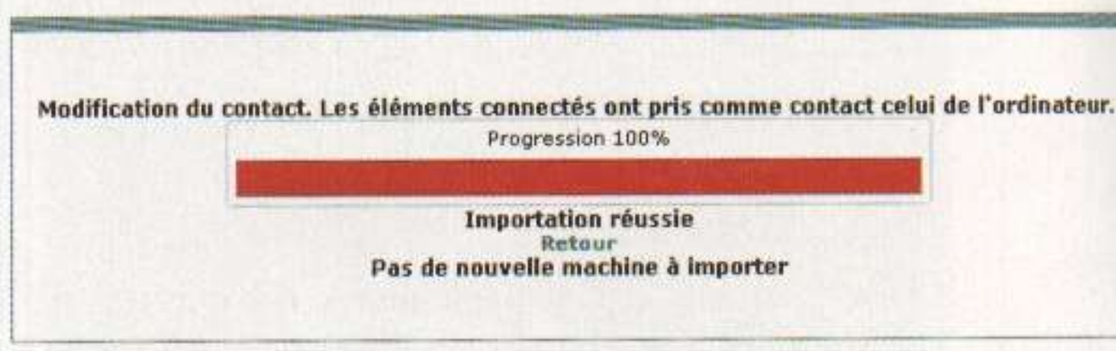
Puisque, dans notre cas, nous avons déjà scanné des machines, il nous suffit d'aller dans *Importation de nouveaux ordinateurs*.



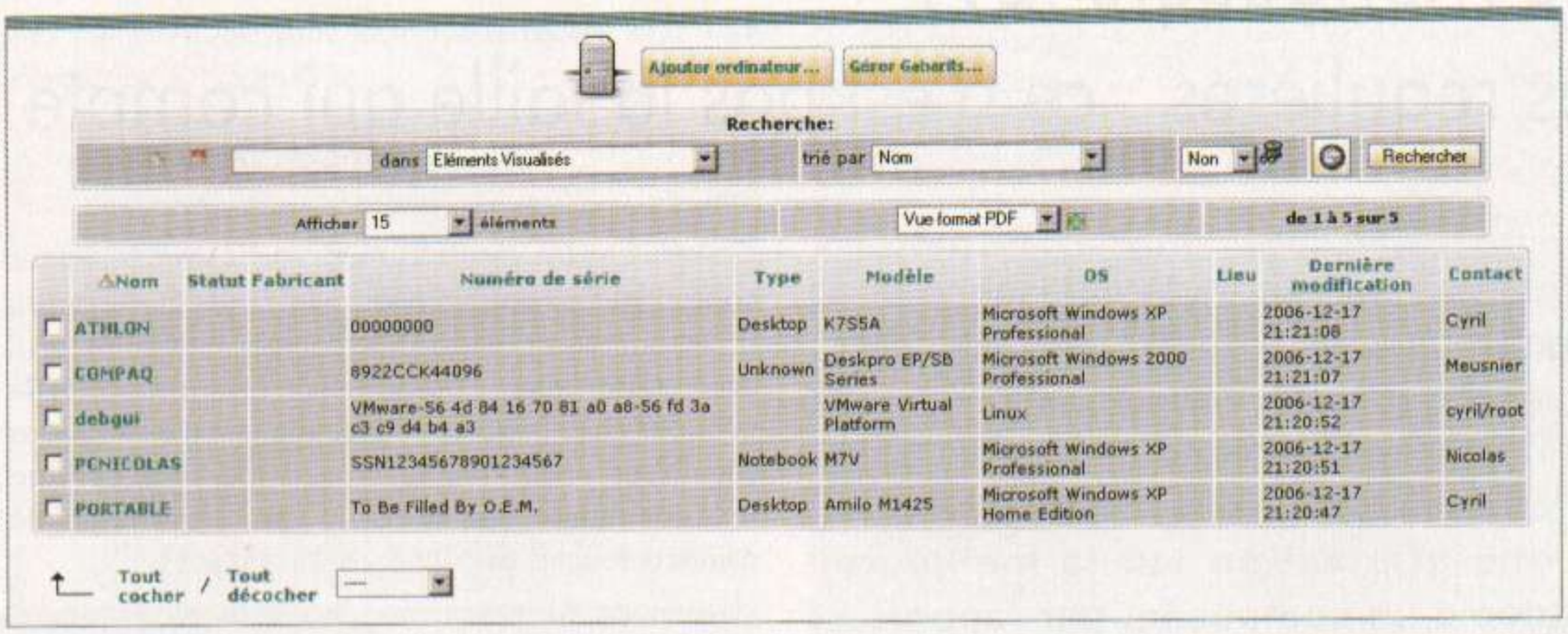
On voit apparaître les ordinateurs existant dans OCS. Il suffit de cocher ensuite ceux que nous désirons récupérer. Une fois validé, une page nous montrera une barre de progression nous indiquant l'avancement des traitements.



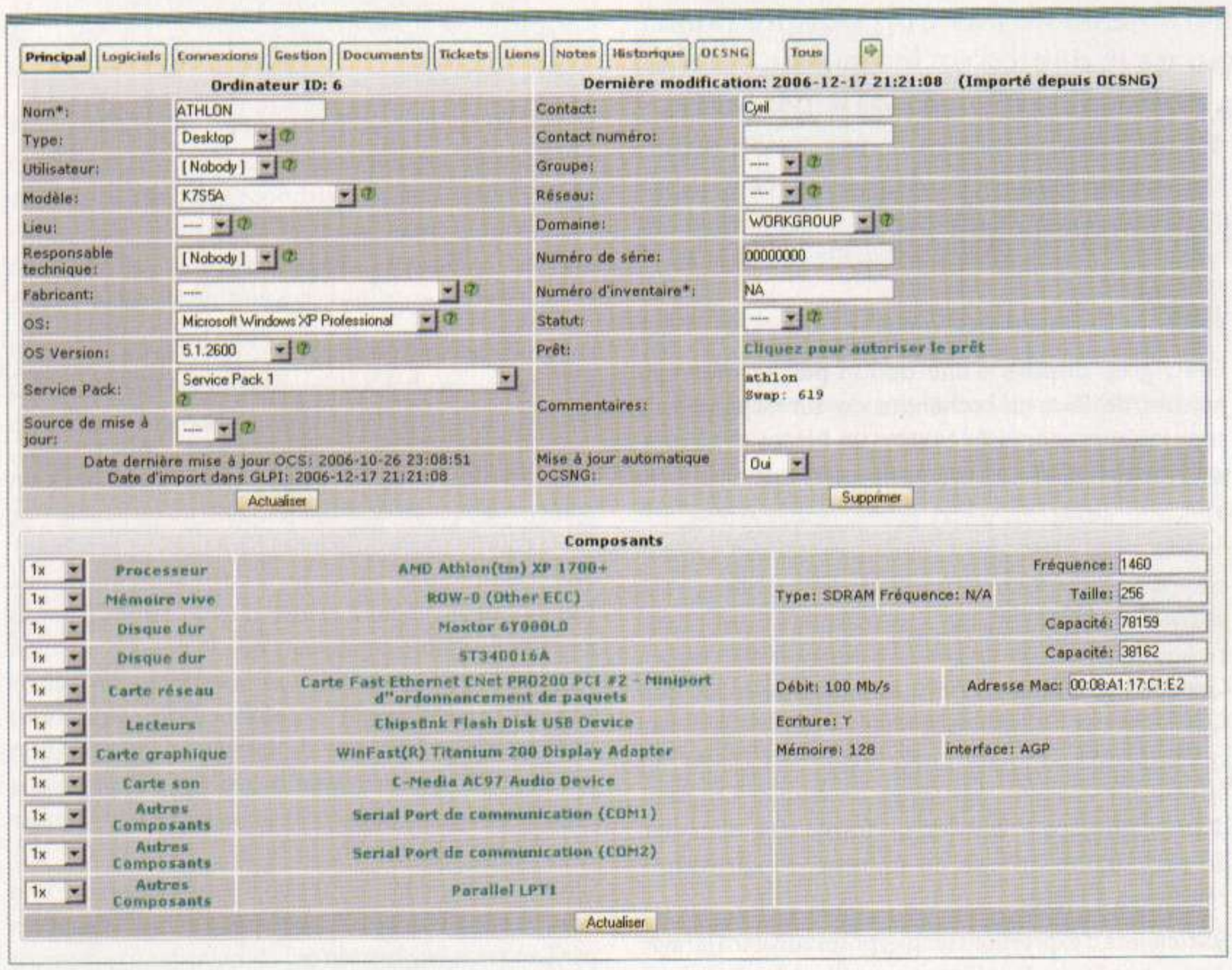
Puis enfin :



Ensuite, allez dans *Inventaire -> ordinateurs* et vous obtiendrez la liste contenue dorénavant dans GLPI.



Il suffit de cliquer sur un nom pour afficher les données propres à une machine.



Comme vous pouvez le voir ici, beaucoup de données sont donc répertoriées, comme le processeur, la RAM, le disque dur, la carte réseau, les lecteurs, la carte graphique. Vraiment tout ce qui se trouve dans la machine est recensé.

Conclusion

OCS est un logiciel *open source* qui permet une gestion efficace de votre parc matériel et logiciel. Son implémentation est simple et n'entraîne que peu de modifications au sein de votre architecture. Couplé avec des bases de donnée performantes, telles que SQL Serveur ou MySQL, vous pourrez effectuer de nombreux recoupements. Si, de plus, vous associez à OCS un utilitaire tel que GLPI, vous pourrez gérer efficacement et simplement l'ensemble de votre parc (aussi bien au niveau matériel/logiciel qu'au niveau utilisation). Quelques bugs existent encore concernant la remontée de périphériques, mais, sinon, l'association des deux applications reste agréable d'utilisation.

Cyril Meusnier,

cyril.meusnier@etu.univ-tours.fr

Philippe « BooK » Bruhat

Perles de Mongueurs (29)

Expressions régulières : ce n'est pas la taille qui compte

À moins d'avoir besoin de la puissance des expressions régulières de Perl, il est légitime de penser que GNU grep sera plus rapide que perl -ne 'print if /regexp/'. Après tout, grep est écrit en C, et Perl est « interprété », n'est-ce pas ? Tant qu'une affirmation sur la meilleure performance d'un composant par rapport à un autre n'a pas été vérifiée par l'expérience, il est préférable de ne pas trop y croire (sauf si on compare le débit d'un lecteur de cartes perforées à celui d'un disque SATA, peut-être). De plus, toute comparaison a des limites de validité. Nous allons donc vérifier dans quels cas grep est meilleur que Perl.

Améliorer les performances de grep

GNU grep dispose d'une option peu connue, qui permet de faire sa recherche de correspondance sur plusieurs motifs listés dans un fichier :

```
    -f FICHIER, --file=FICHIER
          Lire  les motifs dans le FICHIER indiqué,
un motif par ligne. Un
          fichier vide ne contient aucun motif, si
bien qu'aucune  concor-
          dance n'est trouvée.
```

Cependant, il faut prendre en compte le fait que **GNU grep** ne réalise aucune optimisation sur ces expressions : il se contente de les tester en boucle sur chacune des lignes en entrée. On peut donc s'attendre à voir les performances diminuer en proportion inverse du nombre d'expressions régulières à tester.

Soit une liste d'expressions simples prises au hasard (à l'aide du module `Acme::MetaSyntactic` et du thème `unicode`) pour construire un fichier d'expressions régulières :

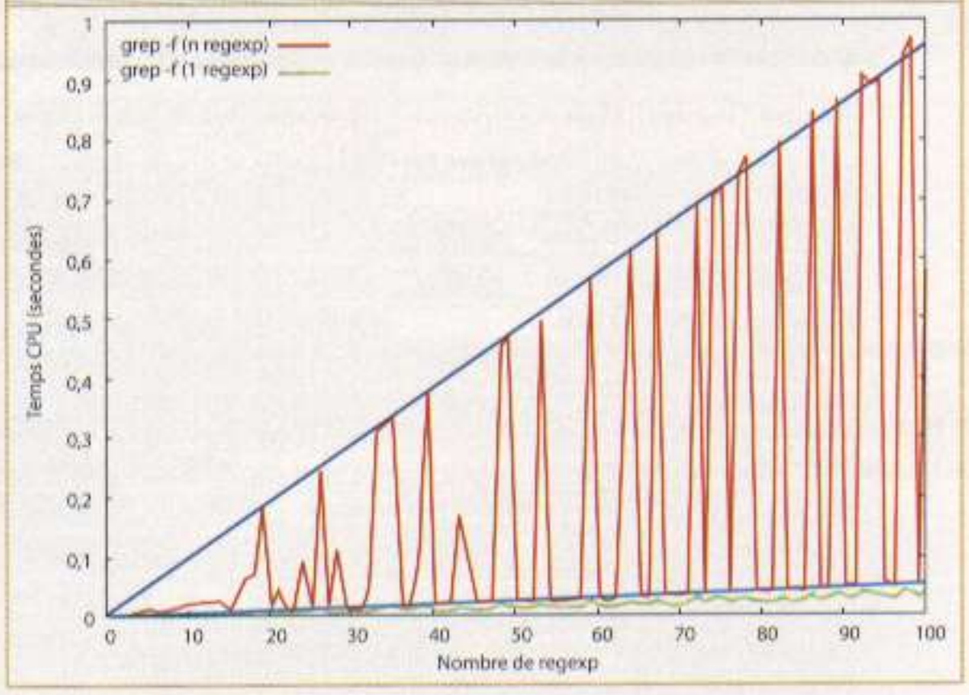
```
AEGEAN_NUMBER_TWENTY
FULLWIDTH_LATIN_CAPITAL_LETTER_S
GREEK_ACROPHONIC_ATTIC_FIFTY_THOUSAND
LATIN_CAPITAL_LETTER_O_WITH_TILDE_AND_ACUTE
MUSICAL_SYMBOL_ORNAMENT_STROKE_5
```

On peut combiner ces cinq expressions en une seule, avec la magie de `\|` :

```
AEGEAN_NUMBER_TWENTY\
|FULLWIDTH_LATIN_CAPITAL_LETTER_S\
|GREEK_ACROPHONIC_ATTIC_FIFTY_THOUSAND\
|LATIN_CAPITAL_LETTER_O_WITH_TILDE_AND_ACUTE\
|MUSICAL_SYMBOL_ORNAMENT_STROKE_5
```

Comparons les résultats de `grep -f` avec le fichier contenant *n* expressions et celui de `grep -f` avec un fichier contenant l'expression combinée. Pour les tests, nous utiliserons une boîte mail de 116 389 octets (la taille du fichier importe peu) et une liste d'expressions régulières produites au hasard (ce sont comme précédemment des noms de caractères Unicode distincts fournis par `Acme::MetaSyntactic`).

Figure 1 : Comparaison des performances de grep



Lors de mes tests, j'ai constaté que **grep** avait un comportement assez erratique (ainsi qu'on peut le voir sur la courbe rouge du graphe). Sans pouvoir le prouver avec certitude, je pense qu'il s'agit d'optimisations (ou de bogues) liées à la liste d'expressions passées à **grep**. Il semble cependant que la courbe est bien comprise entre deux droites, qui correspondraient aux cas le pire (bogue ou absence d'optimisation) et le meilleur (absence de bogue ou optimisation enclenchée). De plus, avec beaucoup d'expressions régulières, j'ai constaté qu'il est assez facile de tomber sur des cas défavorables.

On voit cependant le gain qu'il y a à remplacer une boucle qui passe une liste d'expressions régulières par une seule expression qui combine les expressions de la liste en question : la courbe verte est parfois **très** en dessous de la courbe rouge (d'un facteur 10 ou 20), ce qui fera une différence lorsqu'on traite de gros fichiers de données.

Regexp::Assemble

Lassé du spam et des virus qui engorgeaient son serveur STMP, et constatant que nombre d'entre eux provenaient d'adresses de connexion ADSL chez des particuliers (probablement infestés de virus et de chevaux de Troie), David Landgren a pris des mesures radicales : il a considéré que les messages venant de connexions personnelles (*dialup*) pouvaient être directement rejetés.

Les lignes ADSL sont faciles à détecter avec une résolution DNS inverse :

```
64-80-231-201.fibertel.com.ar
host-67-84-230-24.midco.net
host-89-229-2-176.torun.mm.pl
host-213-213-233-44.brutele.be
ppp-58.10.219.243.revip2.asianet.co.th
68-67-200-36.atlaga.adelphia.net
```

Il est facile d'en déduire quelques expressions régulières pour trouver les utilisateurs du même type :

```
\d+-\d+-\d+-\d+\.fibertel\.com\.ar
host-\d+-\d+-\d+-\d+\.midco\.net
host-\d+-\d+-\d+-\d+\.torun\.mm\.pl
host-\d+-\d+-\d+-\d+\.brutele\.be
ppp-\d+\.\d+\.\d+\.\d+\.revip\d+\.asianet\.co\.th
\d+-\d+-\d+-\d+\.atlaga\.adelphia\.net
```

Postfix (le MTA utilisé par David) peut utiliser la librairie PCRE (*Perl Compatible Regular Expressions*) pour faire des correspondances sur de tels noms de domaine à rejeter automatiquement. La première approche de David a donc consisté à produire une liste d'expressions régulières (comme pour `grep -f`).

Comme nous l'avons vu précédemment, les performances se dégradent rapidement quand on a quelques centaines d'expressions à passer sur chaque connexion entrante. David a donc commencé à écrire un optimiseur d'expressions régulières qui va plus loin que le simple `join '|'`. L'idée est de grouper les parties communes pour optimiser la recherche. Un exemple valant mieux qu'un long discours, voici le résultat sur les six expressions précédentes :

```
(?:
  host-\d+-\d+-\d+-\d+\.
  (?:
    torun\.mm\.pl
    |brutele\.be
    |midco\.net
  )
  |\d+-\d+-\d+-\d+\.
  (?:
    atlaga\.adelphia\.net
    |fibertel\.com\.ar
  )
  |ppp-\d+\.\d+\.\d+\.\d+\.revip\d+\.asianet\.co\.th
)
```

Ce travail a donné lieu à une présentation aux premières « Journées Perl » en 2004. La réaction des auditeurs a été unanime : « Il faut en faire un module CPAN ! ».

C'est ainsi qu'est né le module `Regexp::Assemble`.

Son utilisation est très simple, comme le montre ce petit script, utilisé pour comparer les performances d'un programme Perl basé sur `Regexp::Assemble` avec `grep -f` :

```
#!/usr/bin/perl
use strict;
use warnings;
use Regexp::Assemble;

my $ra = Regexp::Assemble->new();
{
    local @ARGV = shift;    # joli hack
    # ajout des expressions à la structure
    $ra->add($_) while <>;
}

# récupération de l'expression optimisée et
compilée
my $re = $ra->re();

# utilisation de l'expression dans un traitement
while (<>) {
    print if /$re/;
}
```

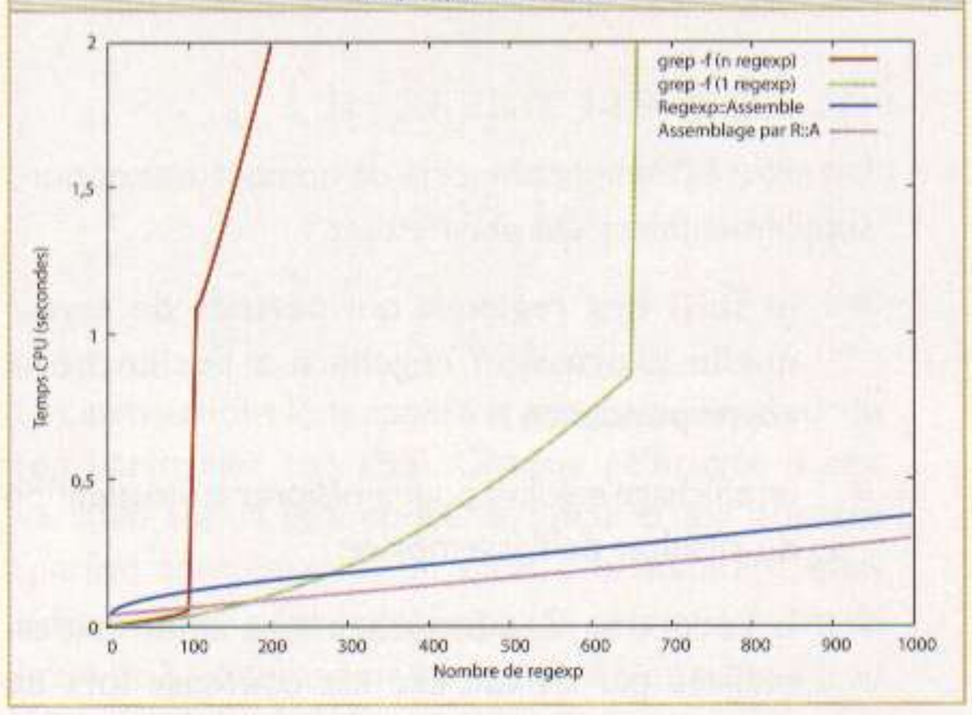
Ce script assemble les expressions contenues dans le premier fichier passé en paramètre (le *hack* mentionné dans les commentaires) et les applique au reste des fichiers passés en paramètre, pour produire un résultat équivalent à `grep -f`.

Comparaison de grep et Regexp::Assemble

Revenons à notre exemple de départ. À partir d'une liste de noms de caractères Unicode, nous prenons les *n* premières pour produire nos listes de test. Ceci nous permet d'éviter les cahots liés aux différences entre les listes de regexps tirées au hasard.

Enfin, le script ci-dessus sera utilisé, ainsi qu'une version modifiée qui se contente de compiler l'expression rationnelle assemblée. Le fichier de test est le même que précédemment.

Figure 2 : Comparaison des performances de grep et Regexp::Assemble



Quelques conclusions sur ce graphique :

1. `grep -f` avec *n* regexps peut rapidement faire augmenter le temps de traitement avec l'ajout d'une expression régulière à la liste. Ce comportement me fait pencher pour un bogue dans le programme.

2. Avec une expression optimisée par simple concaténation avec des \|, les performances de **grep** se dégradent beaucoup moins vite qu'avec une liste et l'option -f.

Mais au bout d'un certain nombre d'expressions, on rencontre à nouveau des problèmes de dégradation des performances.

3. Une expression optimisée par `Regexp::Assemble` permet à un programme Perl de devenir plus rapide que **grep** (pourtant écrit en C et certainement très optimisé). Ceci dépend des expressions à assembler, mais c'est vrai dès quelques centaines d'expressions.

4. Plus il y a d'expressions à assembler, plus le temps d'assemblage par `Regexp::Assemble` augmente. En fait, on constate que le temps de traitement du fichier de données par Perl à l'aide de l'expression régulière obtenue est quasiment constant.

5. Il semblerait que le temps d'assemblage augmente linéairement avec le nombre d'expressions à assembler.

Même s'il faut deux secondes et demie sur ma machine pour assembler 10 000 expressions régulières, ce coût d'initialisation est largement compensé par le gain en performance que procure cette optimisation.

De plus, dans des cas comme celui du Postfix de David, l'expression n'a besoin d'être calculée qu'à chaque fois qu'il rajoute des éléments à sa liste noire, le résultat faisant partie de la configuration de Postfix. C'est tout bénéfice lors de l'exécution de Postfix.

La configuration actuelle de David comporte 4096 expressions régulières, soit 136 467 octets bruts et 90 549 octets après assemblage.

Et ce n'est pas tout !

`Regexp::Assemble` comporte de nombreuses options supplémentaires qui permettent :

- le suivi des regexps, qui permet de savoir quelle expression régulière a déclenché la correspondance ;
- un affichage enjolivé pour améliorer la visualisation du résultat de l'assemblage ;
- la réduction des branches, une optimisation réalisée par défaut. Elle est coûteuse lors de l'assemblage des expressions, mais peut être désactivée.

Enfin, le fichier `README` contenu dans la distribution explique l'algorithme utilisé par `Regexp::Assemble`.

Philippe « BooK » Bruhat,

book@mongueurs.net,

de Lyon.pm et Paris.pm

RÉFÉRENCES

Cette perle reprend des éléments de la présentation que David Landgren a faite aux *Journées Perl* 2006. Voici tous les liens utiles :

- La présentation de `Regexp::Assemble` aux *Journées Perl* : **http://conferences.mongueurs.net/fpw2006/slides/regexp-assemble.html**
- Le site Perl de David : **http://www.landgren.net/**
- *Perl Compatible Regular Expressions* (PCRE) : **http://www.pcre.org/**
- `Regexp::Assemble` sur CPAN : **http://search.cpan.org/dist/Regexp-Assemble/**
- `Regex::PreSuf`, un précurseur de `Regex::Assemble` par Jarkko Hietaniemi : **http://search.cpan.org/dist/Regex-PreSuf/**
- `Acme::MetaSyntactic`, le module qui m'a pris deux ans de ma vie : **http://search.cpan.org/dist/Acme-MetaSyntactic/**

À vous !

Envoyez vos perles à **perles@mongueurs.net**. Elles seront peut-être publiées dans un prochain numéro de *Linux Magazine*.

Yann Guidon

GLMF, le magazine que Richard Stalman^WStallmann^WStallman^W^WRMS lit en cachette, présente :

Quelques techniques d'optimisation en C

Programmer est un art, surtout en C où d'innombrables facteurs interviennent : le processeur, l'algorithme à exécuter, le compilateur, l'environnement logiciel... Voici quelques techniques illustrées qui vous permettront de comprendre mon style de codage très particulier.

Malgré leur sophistication théorique, les bouts de code que j'écris tournent souvent autour de traitements itératifs très simples sur un petit tableau. L'impératif de vitesse oblige à utiliser des formes de codage particulières, qui permettent d'améliorer l'efficacité du code généré par le compilateur. Ces techniques vont peut-être à l'encontre du style établi, mais évitent d'avoir recours au langage assembleur, ce qui augmente la portabilité des logiciels.

1. GCC et ses petites manies

Commençons par un code d'exemple inutile (il est évident qu'il ne sert à rien en l'état), mais qui illustre très bien le propos, car de nombreux ingrédients typiques sont réunis :

```
int fonction_stupide(int x) {
  int table[4];
  int i, j=0;
  /* initialisation */
  for (i=0; i<4; i++) {
    table[i]=x+i+1;
  }
  do {  /* Boucle dans le vide */
    for (i=0; i<4; i++) {
      j^=table[i]; /* une opération simple quelconque */
    }
  } while (x--);
  return j;
}
```

La remarque la plus importante est qu'il est devenu compliqué, avec les raffinements progressifs de GCC, de lui faire générer du code inutile, car ce dernier est simplement supprimé par l'optimiseur. A part cela, le code source ne suscite aucun commentaire, sauf si on le regarde sous l'angle de l'efficacité. Voici du code généré par `gcc 3.3.4` :

```
; Code assembleur lourd et disgrâcieux généré
; par "gcc -S exemple.c". Attention : ne pas le montrer
; à des cardiaques ou à des mineurs de moins de 25 ans
fonction_stupide:
; Prologue
        pushl   %ebp            ; sauve la trame précédente
        movl    %esp, %ebp
      ; puis alloue 40 octets pour la trame
        subl    $40, %esp
        movl    $0, -32(%ebp)  ; j=0
        movl    $0, -28(%ebp)  ; i=0
; Exemple de boucle pas du tout optimisée :
.L2:
      ; si i<3, aller à L5 sinon aller à L3
        cmpl    $3, -28(%ebp)
        jle     .L5
        jmp     .L3 ; nb: inverser la condition
                ; aurait économisé une
                ; instruction et un saut
.L5:
        movl    -28(%ebp), %edx ; edx=i
        movl    -28(%ebp), %eax ; eax=i  (pourquoi dupliquer ?)
        addl    8(%ebp), %eax   ; eax+=x (paramètre sur la pile)
        incl    %eax            ; eax++
        movl    %eax, -24(%ebp,%edx,4)  ; tableau[i]=i+x+1
        leal    -28(%ebp), %eax
        incl    (%eax)          ; i++
    /* fin du corps de la boucle d'initialisation */
        jmp     .L2
.L3:
        nop     /* là, ne fait vraiment rien :-/ */
.L6:
        movl    $0, -28(%ebp)   /* i=0 */
/* Boucle à structure aussi moche que la première : */
.L9:
        cmpl    $3, -28(%ebp)
        jle     .L12
        jmp     .L8
.L12:
        movl    -28(%ebp), %eax         ; eax=i
        movl    -24(%ebp,%eax,4), %edx  ; edx=&tableau[i]
        leal    -32(%ebp), %eax  ; eax=&j
        xorl    %edx, (%eax)     ; *eac^=edx : j^=tableau[i]
        leal    -28(%ebp), %eax  ; eax=&i
        incl    (%eax)           ; i++
        jmp     .L9  ; fin du corps de la boucle,
                ; retourne à la structure de décision
.L8:
        decl    8(%ebp)  ; x-- (dans le while)
        cmpl    $-1, 8(%ebp)
        jne     .L6      ; si x>=0, reboucle encore une fois
        movl    -32(%ebp), %eax  ; return j (dans eax)
; épilogue :
        leave   ; restaure la trame de pile précédente
        ret     ; fin de la fonction
```

Les instructions font appel à la pile, ce qui les alourdit (en particulier sur x86). Chaque référence à une variable sur la pile coûte le calcul d'une adresse (parfois complexe) et un accès à la mémoire. Mais surtout, la pile est pointée par le *pointeur de trame* (`ebp`) et lui-même doit être initialisé et restauré au début et à la fin de la fonction (les fameux prologues et épilogues). Pourtant, `ebp` n'est pas réservé uniquement à cet usage et pourrait servir à autre chose (pour les calculs, donc) si on le libérait. Et comme `ebp` est calculé à partir de `esp` (le pointeur de pile) qui n'est pas utilisé, on perd vraiment un registre !

2. Il suffit pourtant de demander gentiment

Si on utilise un nombre limité de variables locales, dans le cadre d'une boucle déroulée, le compilateur associera un registre à chaque variable, ce qui élimine d'un coup toute question d'adressage. Pour cela, il faut faire appel à des options de compilation spéciales :

```
gcc -Os -fomit-frame-pointer -momit-leaf-frame-pointer \
-march=pentium3 exemple.c
```

Le plus important est `-Os` qui permet l'accès à d'autres registres inutilisés auparavant (sur x86, qui concerne la majorité des lecteurs) : `ebx`, `ecx`, `edi`, `esi`. Si on ajoute `eax`, `edx` et `ebp`, nous pouvons maintenant faire tenir 7 variables dans le cœur du processeur et y accéder instantanément.

Dans du code déroulé, les inférences de valeurs permettent encore de réduire le nombre d'instructions exécutées. Le code généré reste habituellement plus long, mais son exécution est beaucoup plus rapide ! Voici un exemple de code C fonctionnellement identique au précédent, mais plus efficace :

```
int fonction_stupide_un_peu_plus_rapide(int x) {
  int j=0,
    table1=x+1, table2=x+2, table3=x+3, table4=x+4;

  do {
    j ^= table1^table2^table3^table4;
  } while (x--);

  return j;
}
```

Combiné avec les options de compilation décrites plus haut, le code est non seulement plus rapide, mais paradoxalement aussi plus court, malgré le prologue et l'épilogue plus longs, ceci grâce aux nombreux calculs d'adresses économisés :

```
; Code assembleur assez bon, généré par
; gcc -Os -fomit-frame-pointer -momit-leaf-frame-pointer
;       -march=pentium3 -S exemple.c

fonction_stupide_un_peu_plus_rapide:
; Prologue : empile/sauve ebp, edi, esi et ebx
        pushl   %ebp
        xorl    %ecx, %ecx ; ecx=j=0
        pushl   %edi
        pushl   %esi
        pushl   %ebx
; Initialisation :
        movl    20(%esp), %edx ; edx=x
        leal    1(%edx), %ebp  ; ebp=x+1=tableau1
        leal    2(%edx), %edi  ; edi=x+2=tableau2
        leal    3(%edx), %esi  ; esi=x+3=tableau3
        leal    4(%edx), %ebx  ; ebx=x+4=tableau4
; Boucle principale :
.L20:
        movl    %ebp, %eax
        decl    %edx        ; x-- (dans le while)
        xorl    %edi, %eax  ; eax=tableau1^tableau2
        xorl    %esi, %eax  ; eax^=tableau3
        xorl    %ebx, %eax  ; eax^=tableau4
        xorl    %eax, %ecx  ; j^=table1^table2^table3^table4;
        cmpl    $-1, %edx
        jne     .L20    ; reboucle si x>=0
; Epilogue : dépile tous les registres sauvés
        popl    %ebx
        movl    %ecx, %eax ; retourne j dans eax
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

Le seul point qui peut être encore optimisé est l'utilisation de `eax` comme variable temporaire dans la boucle. Le xor peut être réalisé directement avec `ecx` (j) grâce à la propriété de commutativité (ce que le compilateur ignore). Une écriture différente du code source remédie à cela, ce qui libère un registre qui peut maintenant être associé à une variable supplémentaire (ici, on augmente la taille du tableau).

```
int autre_fonction_stupide(int x) {
  int j=0,
    table1=x+1, table2=x+2, table3=x+3,
    table4=x+4, table5=x+5;

  do {
    j^=table1;
    j^=table2;
    j^=table3;
    j^=table4;
    j^=table5;
  } while (x--);

  return j;
}
```

Le code final contient 7 variables (table, j et x) et tient entièrement dans les 7 registres d'un x86, ce qui le rend à la fois rapide et concis.

3. Recommandations

Nous en déduisons les conseils suivants, qui ne sont pas spécifiques aux processeurs x86 :

- Ne pas hésiter à aller **fouiner dans la sortie du compilateur** et dans les nombreuses ressources fournies aux développeurs. On économise parfois un ticket de cinéma pour le dernier film d'horreur, tout en améliorant la qualité du code source.
- En général, la stratégie consiste à **écrire du code C qui peut être traduit directement en langage machine**, en réduisant l'abstraction et en explicitant les formats ou les structures. Il faut qu'il y ait une **correspondance directe entre ce qu'on demande au processeur et ses instructions**. Pour que cela soit efficace, il faut bien connaître la machine cible et ce qu'elle fait le mieux. Cela se limite souvent aux quelques fonctions de base disponibles, telles que l'addition, la soustraction et quelques autres fonctions logiques, mais cela varie énormément d'une machine à l'autre !
- Le premier examen du travail de GCC sans optimisation montre à quel point la gestion des boucles peut être déficiente. Pour la plupart des boucles simples, je privilégie la structure `init; do {X; inc;} while (cond)` qui est équivalente à `for (init;cond;inc) {X}`. La gestion des variables est

manuelle, ce qui réduit un peu la lisibilité, mais la traduction manuelle en code assembleur est directe, surtout si la condition de rebouclage est réduite à un test de base (variable égale à zéro ou négative).
Si le corps de la boucle devait être sauté (condition initialement invalide), un simple `if (cond) {do {X} while (cond)}` suffit, évitant d'avoir à utiliser la forme `while (cond) {X}`, équivalente mais traduite de manière un peu plus complexe. La duplication du test dans `if (cond)` peut être encombrante, mais cela pose rarement un souci.

- Sur x86 et autres *architectures à deux adresses*, **découper les expressions complexes et utiliser la syntaxe à deux adresses**. Comme cette syntaxe se traduit souvent directement en une instruction, cela réduit la pression sur l'allocation des registres (lorsque c'est correctement utilisé).

```
a = a + b; /* syntaxe classique */
a += b;    /* syntaxe à deux adresses */
```

- Ne pas hésiter à **forcer l'ordre des opérations**. Le compilateur obéira souvent sans discuter (mais il faut le vérifier dans le code généré) et cela peut débloquer quelques situations tendues (comme nous venons de le voir).
Le compilateur ne tentera pas d'exploiter des propriétés comme la commutativité ou la distributivité des opérateurs, car le standard du langage C exige de conserver l'ordre des opérations. C'est parfois capital pour les entiers, à cause d'éventuels dépassements de capacité sur des variables temporaires, mais c'est surtout critique pour les nombres à virgule flottante, car le résultat dépend de l'ordre des opérandes.
- **Un petit tableau est réalisé à partir de variables scalaires locales** pour que le compilateur puisse assigner chaque entrée à un registre et ainsi minimiser les mouvements de données et les accès à la mémoire.
- Autant que possible, **utiliser des variables statiques globales** (déclarées hors de la fonction). D'une part, cela réduit au minimum l'occupation de la pile, ce qui peut même éviter d'y accéder complètement dans le meilleur des cas, donc de modifier des pointeurs (avec l'option `-fomit-frame-pointer` de GCC). D'autre part, les modes d'adressage complexes sont plus lents à décoder (sur x86) ou longs à coder (avec les processeurs RISC). Dans un programme, le mieux est d'utiliser uniquement un registre pointeur avec éventuellement un déplacement (*offset*) constant. Par exemple :

```
int t[1337];
int *p=&t;  /* création d'un alias */
int h=t[0]  =*p;      /* OK, accès direct par pointeur */
int i=t[10] =*(p+10); /* OK, un pointeur plus un offset constant */
int j=t[i]  =*(p+i);  /* Délicat, offset non constant, donc
                         mode d'adressage un peu moins rapide */
int k=t[i*281] /* Encore pire, il faut calculer l'offset */
```

On comprend alors l'intérêt de mettre des variables en global (**sauf si la fonction doit être récursive ou réentrante**) : une variable globale a une adresse constante (donc adressée directement), alors qu'une variable sur la pile est accédée au travers du pointeur de pile (`esp`) ou du pointeur de trame (`ebp`), cette indirection peut ralentir les accès. *Attention : ce n'est pas une science exacte, mais une indication de l'effort que le microprocesseur devra fournir.*

- En conséquence, si une variable n'est pas accédée souvent, il faut la mettre **dans une variable globale**. Cela empêche le compilateur de l'associer à un registre, ce qui laisse plus de place pour les autres variables locales. Et comme l'accès est presque immédiat grâce à l'adressage direct, le surcoût est minimal.
- **La boucle est déroulée autant de fois qu'il y a de registres**. Cela réduit l'importance relative du comptage des itérations (libérant alors un registre qui aurait servi comme compteur de boucle) et permet de nombreuses autres simplifications par inférences de valeur entre plusieurs itérations. Dans le code d'exemple, cela revient à donner directement la valeur au lieu de la recalculer à chaque ligne.
- L'opération, même peu complexe, est **codée dans une macro**, qui réduit la taille du code source et les chances d'erreurs de copier-coller-modifier. Je ne le répèterai pas assez : les macros sont destinées à nous faciliter la vie, il faut en profiter ! Dans le passé, j'ai perdu un temps incroyable à vouloir tout faire à la main... Le développement est ralenti, et le débogage peut devenir absolument impossible !
- Les paramètres de la macro incluent les indices du tableau à accéder. Comme les indices sont **codés en dur** au moment de la compilation, le processeur ne perd plus un cycle à les calculer lors de l'exécution, ce qui est bienvenu lors d'accès circulaires par exemple.

L'optimisation du code doit se faire impérativement APRES étude du meilleur algorithme. N'utilisez pas les conseils donnés ici comme des règles de développement.

4. Ordonnancement des instructions

Il n'est pas possible d'exposer toutes les techniques d'optimisation existantes, mais je dois encore évoquer un dernier point important : l'ordre des instructions dans un programme et leur réordonnancement dynamique dans les processeurs à *exécution dans le désordre* (*Out Of Order*, ou **OOO**, mais rien à voir avec *OpenOffice.Org*). Cela concerne les Pentium2/3/4, la lignée des PowerPC ou les Alpha21264 (EV6 et plus).

Les processeurs superscalaires des générations précédentes (tels les premiers Pentium, MIPS, SPARC ou Alpha21164) demandent au compilateur de fabriquer un flux d'instructions finement calibré, capable d'utiliser au mieux les ressources cycle par cycle, avec des règles parfois complexes et contradictoires (ah ! l'optimisation pour le PentiumMMX...). Par contre,

les processeurs **OOO** peuvent exécuter un certain nombre d'instructions sans attendre le résultat des précédentes, s'il n'y a pas de dépendances entre elles. Ces processeurs sont bien plus complexes, mais les gains de temps sont substantiels dans certaines circonstances, en particulier pour compenser la latence des accès à la mémoire.

Afin d'améliorer un programme, nous pouvons par exemple déplacer (anticiper) une opération jusqu'à 20 instructions à l'avance (selon le processeur et en fonction de la latence). Typiquement, une instruction de lecture en mémoire nécessite plusieurs cycles pour fournir un résultat, lorsque la donnée est déjà en mémoire cache, et le processeur est arrêté pendant des dizaines de cycles s'il faut chercher la donnée en mémoire dynamique externe. Il est donc désirable de reporter les opérations de lecture le plus en amont possible, en fonction des instructions indépendantes que nous pouvons intercaler :

```
int a, b, c, d, i, tableau[];
/* code initial : */
a = tableau[i]+(b*c);
/* code réorganisé */
a = tableau[i]; /* cette instruction prend du temps */
d = b * c;  /* la multiplication est assez longue, aussi. */
/* insérer ici une bonne douzaine d'instructions
qui n'ont rien à voir, pendant que la multiplication et
la lecture s'effectuent en parallèle. */
a += d; /* a et d sont utilisés ici sans blocage */
```

5. Le bon côté du renommage des registres

Comme un processeur OOO exécute les opérations *lorsque les données sont prêtes*, indépendamment de leur ordre dans le programme, on peut traiter un programme par petits blocs indépendants de 3 à 10 opérations. Les processeurs OOO apportent un certain confort lors du codage : le renommage dynamique des registres rend possible la réutilisation à outrance des variables, pour des usages simultanés et différents. C'est critique pour les processeurs x86 qui, nous l'avons vu au début de l'article, manquent cruellement de registres. Depuis le PentiumPro, cette limitation peut être outrepassée dans une certaine mesure.

Si vous avez suivi l'explication du cadre, un nouveau registre renommé est alloué pour contenir le résultat de chaque opération. En termes de code source, cela s'exprime par :

NOTE

Typiquement, un processeur OOO est constitué de plusieurs parties découplées, effectuant leur travail indépendamment, afin de maximiser le parallélisme et de réduire le temps de traitement total.

* **La fenêtre d'instructions** est une mémoire qui communique avec toutes les autres unités du processeur. Elle a une capacité de quelques dizaines de cellules qui contiennent une version modifiée des instructions décodées. Chaque cellule est aussi marquée pour indiquer son état courant : (a) pas d'instruction, (b) instruction décodée en attente d'exécution, (c) en cours d'exécution, ou (d) exécutée.

* **Les registres renommés** sont une autre mémoire, contenant les valeurs temporaires (ou spéculatives) des registres normaux. En plus de la valeur courante, chaque registre renommé indique à quel registre normal sa valeur correspond, et s'il est occupé ou libre.

* **Le décodeur d'instructions** analyse et traduit le flux du programme. Chaque instruction est décomposée en plusieurs parties, ou *champs* : l'opération à effectuer (addition, multiplication, lecture de la mémoire...), le registre destination (qui contiendra le résultat de l'opération) et les registres sources (les opérandes qui doivent être lues).

Lorsque le décodeur ajoute une instruction dans la *fenêtre*, il *traduit* les numéros des registres, ou les *renomme* : il alloue un nouveau registre renommé pour la destination, et cherche à quels autres *registres renommés* correspondent les registres sources. Il utilise une table spéciale pour effectuer ces traductions, qui servira plus tard lors de la validation de l'instruction.

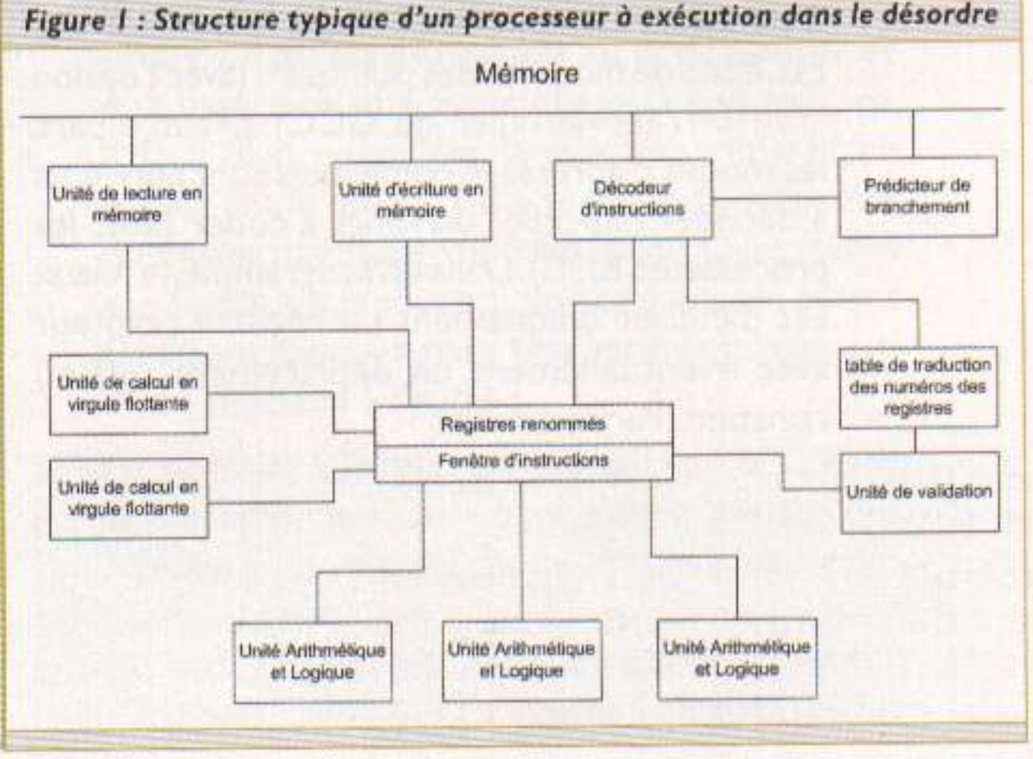
* **Les unités d'exécution** se chargent d'effectuer les opérations telles que l'écriture ou la lecture en mémoire, les calculs sur les nombres entiers ou flottants... Chaque unité, lorsqu'elle n'est pas bloquée par des opérations en cours, scrute la fenêtre d'instructions, à la recherche d'opérations qu'elle est capable d'effectuer (par exemple, l'ALU va chercher uniquement les instructions arithmétiques ou logiques). Il faut aussi vérifier que les opérandes sont prêtes (déjà calculées et disponibles).

Si toutes les conditions sont remplies, les opérandes sont lues dans les registres renommés, l'opération est effectuée, puis le résultat est écrit dans le registre renommé (déjà alloué lors du décodage). Toutes ces opérations nécessitent un nombre variable de cycles, que l'on appelle la *latence*.

* **L'unité de validation** (pour les processeurs Intel, aussi appelée *retrait* ou *commit*). Certains processeurs mettent à jour directement la table de traduction des registres, lorsque le résultat est écrit dans les registres renommés. Pour les processeurs PentiumPro et leurs héritiers (Pentium 2 et 3), une unité spéciale se charge de nettoyer la fenêtre d'instructions, de synchroniser toutes les tables et d'assurer que les instructions soient validées dans l'ordre initial du programme.

Ainsi, le résultat d'une opération peut être prêt avant qu'une instruction qui la précède ne soit terminée.

Figure 1 : Structure typique d'un processeur à exécution dans le désordre



```
int a, b;
/* code source */
a += b;
/* ce que le processeur effectuera vraiment : */
a' = a + b;
```

La variable a' n'existe qu'à l'intérieur du processeur et n'a pas besoin d'être déclarée dans le code source. a' deviendra a lorsque cette dernière sera lue, mais, pendant une période de quelques cycles, les deux cohabitent !

A chaque réécriture du registre, son usage change. Ainsi, si on utilise a plusieurs fois consécutivement, le processeur va effectuer chaque opération en parallèle, sur des versions différentes et simultanées de la variable. Par exemple, imaginons un morceau de code appelant `memcpy()` pour recopier un petit bloc de données.

```
int *a, *b, t;
/* code initial */
memcpy(b,a,32);
/* code déroulé manuellement */
t=a[0]; b[0]=t;
t=a[1]; b[1]=t;
t=a[2]; b[2]=t;
t=a[3]; b[3]=t;
t=a[4]; b[4]=t;
t=a[5]; b[5]=t;
t=a[6]; b[6]=t;
t=a[7]; b[7]=t;
```

Le code déroulé n'utilise que trois registres, mais t pourra exister en huit versions simultanées différentes. Un processeur RISC classique (sans renommage) nécessiterait par contre l'allocation de 4 ou 8 registres différents pour exécuter ce code sans peine.

Mais ce code est assez brutal : il va saturer l'unité d'accès à la mémoire et n'apporte rien par rapport au `memcpy()` initial. Cette fonction, même *inlinée*, est *bloquante*, car elle ne permet pas à d'autres instructions de s'exécuter en même temps.

Par contre, dans la version déroulée, nous pouvons *entrelacer* des instructions indépendantes, ou commencer à effectuer des calculs avec certaines valeurs de la table recopiée, dès que celles-ci sont lues et disponibles dans t. Ainsi peut s'opérer la magie : l'unité d'accès à la mémoire tout comme les unités de calcul peuvent tourner à plein régime en même temps, ce qui réduit le temps d'exécution du morceau de code.

```
int a[], b[], m, n, o, t, u;
/* code de recopie, "saupoudré" d'opérations */
t=a[0]; b[0]=t;
u=a[1]; b[1]=u;
m=t;
m+=u;
t=a[2]; b[2]=t;
u=a[3]; b[3]=u;
n=t;
n+=u;
m+=n;
t=a[4]; b[4]=t;
u=a[5]; b[5]=u;
o=t;
o+=u;
t=a[6]; b[6]=t;
u=a[7]; b[7]=u;
n=t;
n+=u;
n+=o;
m+=n;
```

Le code ci-dessus n'est probablement pas le meilleur code possible, mais il illustre bien l'avantage d'un cœur à exécution dans le désordre. La somme de toutes les données est effectuée au moyen d'un arbre binaire, en utilisant juste 5 variables alors que 15 valeurs différentes sont traitées. De plus, si une lecture prend plus de temps que les autres (à cause d'un *cache miss*), toutes les autres sommes intermédiaires peuvent être calculées sans blocage.

Du point de vue du programmeur de haut niveau, il est intéressant de noter que la granularité du saupoudrage ou l'entrelacement n'est pas critique. On peut entrelacer des blocs d'une, deux, trois ou quatre opérations, puisque le cœur va les réorganiser lui-même. On peut ainsi coder par petits blocs logiques, sans se creuser la tête pour tout bien répartir. Au contraire, les processeurs superscalaires classiques nécessitent une compréhension fine du fonctionnement des *pipelines* et le code a souvent tendance à devenir absolument incompréhensible.

Conclusion

Je n'ai abordé ici que quelques aspects, qui sont spécifiques aux morceaux de code que je publie dans ce magazine. Comme les sources doivent être portables et fonctionner *au mieux* sur une grande variété de processeurs, je me cantonne à des techniques de haut niveau, avec un minimum d'hypothèses sur la plate-forme et en considérant le cas le plus défavorable (ce qui revient à viser les x86).

Il faut aussi remarquer que ces techniques s'appliquent surtout à des fonctions *terminales* (ou *leaf* en anglais), qui n'appellent pas d'autre fonction. On peut ainsi utiliser des variables globales réservées, puisqu'elles ne seront pas surécrites par une autre instance de la même fonction, en plein milieu de leur exécution. Mais dans le cadre d'un programme *multithread*, ce manque de réentrance peut devenir catastrophique.

Enfin, n'oubliez pas de toujours mesurer l'effet d'un changement dans votre code : une modification anodine réserve souvent des mauvaises surprises et une optimisation sophistiquée n'est pas une garantie d'accélération.

Yann Guidon,

whygee@f-cpu.org :: http://ygdes.com
Électronique, musique et informatique en folie^Wliberté

LIEN

- D'autres techniques ont été exposées dans l'article « Hacks en C » de Nicolas Boulay (GLMF n°32).

Olivier Corrio

Placement des contrôles dans une fenêtre en C++ à l'aide de wxWidgets

Le troisième article présentait quelques contrôles en montrant un placement statique. Celui-ci détaille deux méthodes pour placer ces contrôles dans une fenêtre et obtenir un fonctionnement dit « moderne ». Ainsi, il ne sera plus nécessaire par la suite de faire le positionnement des contrôles de façon fixe. Il sera fait de façon automatique.

1. Introduction

Après une petite interruption, voici un nouvel article sur cette bibliothèque portable dont la version actuelle est la 2.7. Cette publication présente les méthodes utilisées sur la plupart des générateurs d'interfaces de la bibliothèque `wxWidgets`. Pour information et sans vouloir les nommer, certains outils professionnels ne proposent en standard que le placement fixe des contrôles. Ici, nous allons voir le placement automatique et adaptatif. Le principe est de laisser un élément de niveau supérieur avoir les informations de changement de taille et de distribuer à sa convenance (ou plutôt à celle du programmeur) la place gagnée ou perdue aux différents contrôles. L'avantage est que le concepteur de l'interface n'a pas à se soucier de la taille et de la position du contrôle, il doit juste faire attention aux règles de distribution de la différence de place. Il existe deux méthodes : l'utilisation d'éléments de type « contrainte » ou l'utilisation d'éléments de type « contrôleur de taille ». La seconde méthode bénéficie toutefois de la préférence des utilisateurs et est préconisée par la documentation. L'utilisation des « contraintes » est qualifiée de dépréciée par la documentation, mais elle permet d'obtenir un effet rapide sans devoir jongler avec les « contrôleurs de taille » qui sont réputés plus difficiles à comprendre (mais cet article lèvera normalement cette difficulté). La compilation des programmes se fait avec la ligne de commande suivante : `g++ mon_prog.cpp -o mon_prog` `` `wx-config --libs - -cppflags` ``. Le ` est obtenu par la combinaison des touches [ALT GR]+[7]. Pour vérifier que la bibliothèque est correctement installée, il suffit de taper `wx-config--version`).

2. Utilisation des contraintes : wxLayoutConstraints

2.1 Première utilisation

La première méthode à employer pour faire un placement harmonieux est d'utiliser les événements `wxSizeEvent` par l'utilisation de la macro `EVT_SIZE` pour modifier la taille et la position des objets. Mais cela reste difficile à gérer pour l'utilisateur. C'est pourquoi `wxWidgets` a introduit la notion de contraintes. Celles-ci sont au nombre de 8 : `left` (gauche), `right` (droite), `top` (haut), `bottom` (bas), `width` (largeur), `height` (hauteur), `centreX` (centré en X) et `centreY` (centré en Y). Pour positionner un composant, il faut connaître 4 contraintes par exemple :

- positionnement à gauche, droite, haut et bas sur une fenêtre ;
- positionnement centré en X, largeur 80%, en haut à 10 pixels et hauteur à 10 pixels ;
- positionnement à gauche, largeur de 150, en bas à 10 pixels, et haut à 10 pixels.

Bref, il faut 4 contraintes sur les 8 possibles et pas une de plus, sinon le résultat est étrange. Pour illustrer ceci, voyons un exemple simple : on souhaite avoir l'écran de la figure 1 et figure 2.

Figure 1 : Fenêtre auto-adaptative configuration 1

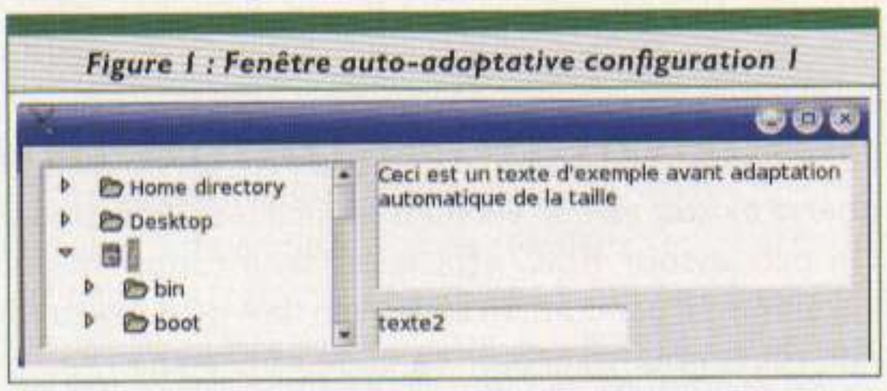
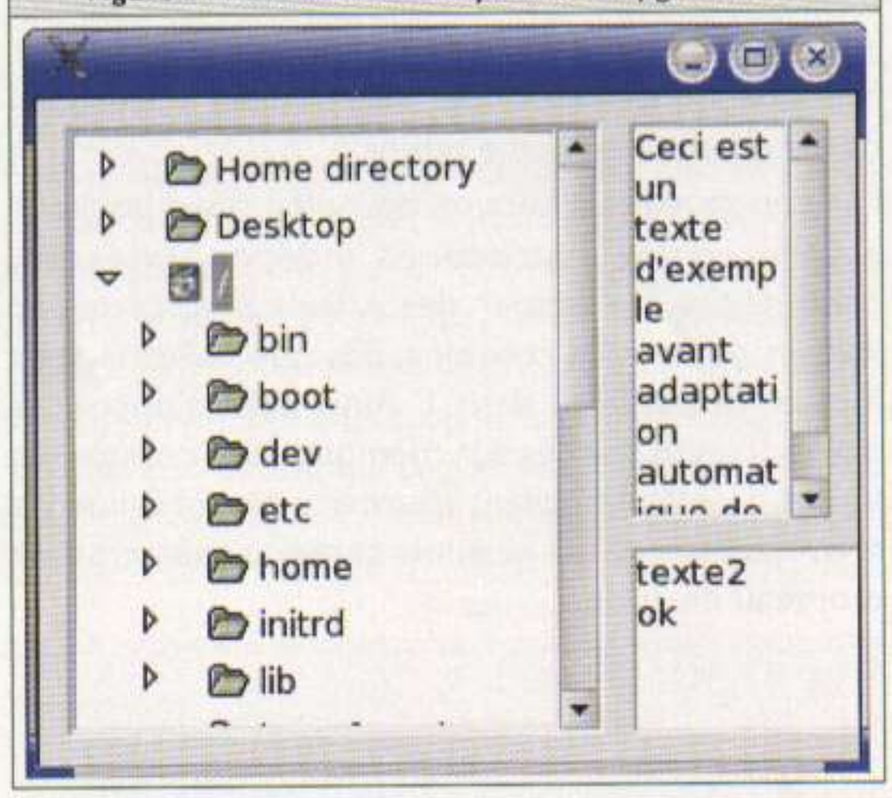


Figure 2 : Fenêtre auto-adaptative configuration 2



Le positionnement du sélecteur de fichier (ou de répertoire) a une taille fixe de 150 pixels et les deux zones de texte sont positionnées à droite du sélecteur, ou ce qui revient (presque) au même, la contrainte gauche des zones de texte est à droite de la contrainte droite du sélecteur. Donc, on a pour les différents éléments les contraintes suivantes (uniquement 4) :

- sélecteur de fichier : contraintes sur la fenêtre uniquement, à gauche de celle-ci, en haut, et bas

avec un écart de 10 pixels et une largeur fixe de 150 pixels ;

- zone de texte du haut : contrainte gauche à 10 pixels de la contrainte droite du sélecteur, en haut de la fenêtre à 10 pixels, même chose pour la droite et 60% de la hauteur de la fenêtre ;
- zone de texte du bas : contrainte à gauche du sélecteur, en bas de 10 pixels sur la fenêtre, en haut à 10 pixels de la contrainte basse de la zone de texte du dessus et largeur de 30% de la fenêtre.

Il est à noter que le fait de positionner une taille de 30% de la fenêtre pour la zone de texte du bas est risqué, car le calcul ne tient pas compte du sélecteur. Il suffit de mettre une taille de 60% et, en changeant légèrement la taille de la fenêtre, une partie de la zone n'est plus visible. Il faut donc bien réfléchir avant d'utiliser les contraintes. Le code complet est alors celui du cadre code suivant :

```
#include <wx/wx.h>
#include <wx/dirctrl.h> // obligatoire car pas contenu dans wx.h

class MyApp : public wxApp // définition de l'application
{
  public:
   virtual bool OnInit(); // la fonction " main " ou Winmain de windows
};

IMPLEMENT_APP(MyApp) // déclaration de l'application

bool MyApp::OnInit() // le " main " de l'application
{
// déclaration d'une fenêtre
  wxFrame *frame = new wxFrame(NULL,wxID_ANY,wxT(""), wxPoint(50,50),
wxSize(450,340));

  wxTextCtrl *text_ctrl1 = new wxTextCtrl(frame, wxID_ANY, wxT("Ceci
est un texte  d'exemple avant adaptation automatique de la
taille"),wxPoint(wxID_ANY, wxID_ANY), wxSize(wxID_ANY, wxID_ANY),wxTE_
MULTILINE); // la première zone de texte
  wxTextCtrl *text_ctrl2 = new wxTextCtrl(frame, -1, wxT("texte2"),wxP
oint(wxID_ANY, wxID_ANY), wxSize(wxID_ANY, wxID_ANY),wxTE_MULTILINE);
// la seconde
// le sélecteur de fichiers
  wxGenericDirCtrl *dir_ctrl= new wxGenericDirCtrl(frame, wxID_ANY);

  wxLayoutConstraints *c1 = new wxLayoutConstraints; // déclaration
                                                        d'une contrainte
  c1->left.SameAs(frame, wxLeft,10); // à gauche de la fenêtre à 10
                                       pixels du bord
  c1->width.Absolute(200); // taille de 200
  c1->top.SameAs(frame, wxTop, 10); // en haut de la fenêtre à 10
                                      pixels du bord
  c1->bottom.SameAs(frame, wxBottom, 10); // idem mais en en bas
  dir_ctrl->SetConstraints(c1); // on fixe la contrainte

  wxLayoutConstraints *c2 = new wxLayoutConstraints;
  c2->top.SameAs        (frame, wxTop,10);
  c2->height.PercentOf  (frame,wxHeight,60); // on se fixe 60% de la
                                               fenêtre => risqué
  // contrainte gauche = droite de la contrainte droite du sélecteur
     avec écart de 10
  c2->left.SameAs       (dir_ctrl, wxRight,10);
  c2->right.SameAs      (frame, wxRight,10);
  text_ctrl1->SetConstraints(c2);

  wxLayoutConstraints *c3 = new wxLayoutConstraints;
  c3->width.PercentOf   (frame,wxWidth,30);
  c3->top.Below         (text_ctrl1,10);
  c3->left.SameAs       (dir_ctrl, wxRight,10);
  c3->bottom.SameAs     (frame, wxBottom,10);
  text_ctrl2->SetConstraints(c3);

// ici on indique un calcul automatique si changement de taille
  frame->SetAutoLayout(true);

  frame->Show(true); // on affiche la fenêtre
  SetTopWindow(frame); // fenêtre principale de l'application
// tout s'est bien passé on continue dans la boucle d'événements
  return true;
}
```

2.2 Seconde utilisation : « tirer » sur la cible

Avant de passer à la partie sur les *sizers*, voici un petit exemple, certes très simple au niveau du fonctionnement, mais qui illustre le fonctionnement des `wxLayoutConstraints`. Pour information, cette méthode est présentée, car elle n'est pas possible avec les sizers, puisqu'il ne peut y avoir d'éléments superposés avec ceux-ci. Voici les photos du jeu figure 3 et figure 4. Même si cela peut paraître étrange avec des boutons, cela pourrait sembler plus réaliste avec des contrôles graphiques créés par l'utilisateur. Il faut changer la fonction `OnInit`, et ajouter un nouveau contrôleur d'événements sur la fenêtre en utilisant un héritage : on a alors le programme source dans les codes suivants.

Figure 3 : 9 boutons superposés

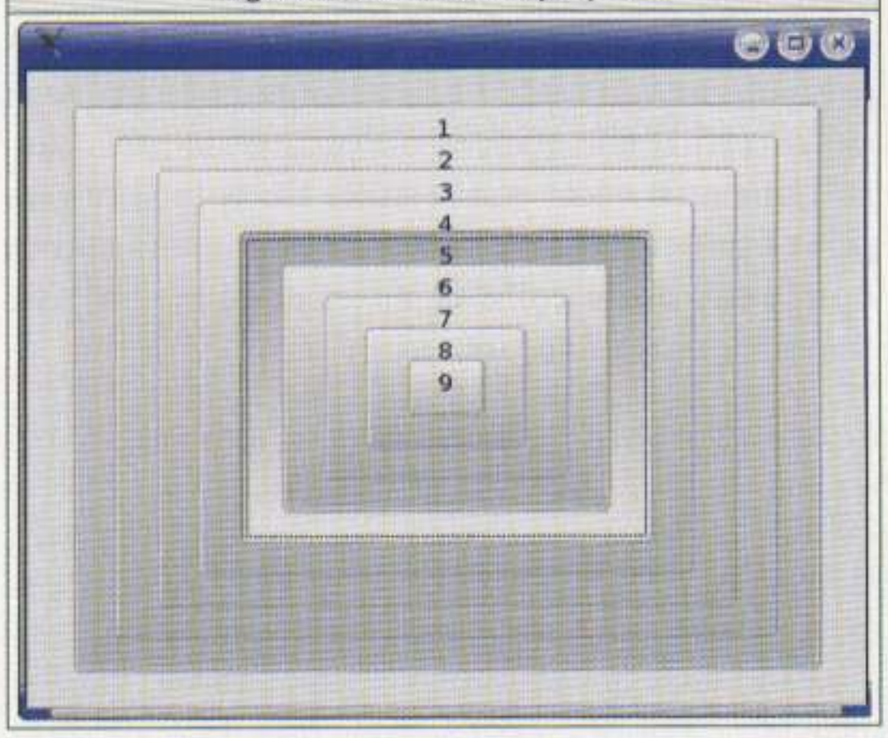
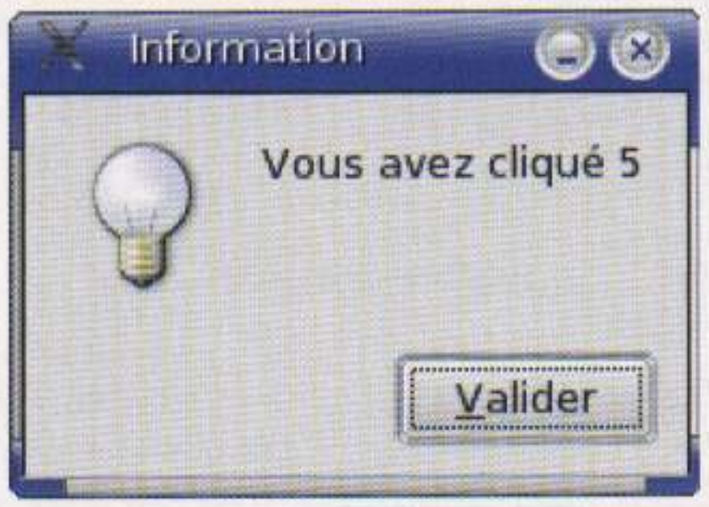


Figure 4 : Une boîte d'information



```
class MyFrame : public wxFrame
{
  public:
   MyFrame(wxWindow *parent,wxWindowID id,wxString text,wxPoint p,wxSize s);
// méthode de gestion du clic
   void OnButton(wxCommandEvent& event);
// déclaration d'une table d'événements
   DECLARE_EVENT_TABLE()
};

// définition de la table d'événements
  BEGIN_EVENT_TABLE(MyFrame,wxFrame)
// on récupère le clic
  EVT_BUTTON(wxID_ANY,MyFrame::OnButton)
END_EVENT_TABLE()

MyFrame::MyFrame(wxWindow *parent,wxWindowID id,wxString text,wxPoint p,wxSize s) :
        wxFrame(parent,id,text,p,s)
{
}

void MyFrame::OnButton(wxCommandEvent& event)
{
  wxButton* button;
// récupération de l'objet émetteur
  button=dynamic_cast<wxButton*>(event.GetEventObject());
  if (button!=NULL) wxMessageBox(wxT("Vous avez cliqué ")+button->GetLabel());
}
```

Le code ci-dessus présente un nouveau type de fenêtre avec un contrôleur d'événements qui vient se placer sur celui de la classe de base ici `wxFrame`. Dans la fonction de récupération de l'événement « clic bouton », on récupère l'objet émetteur. Le *cast* permet de s'assurer qu'il s'agit bien d'un objet bouton (normalement, il ne devrait pas y avoir de souci à ce niveau, mais il est préférable de prendre ses précautions). Une fois le bouton émetteur récupéré, on affiche par une boîte de dialogue le label de celui-ci. Il faut aussi noter que la **taille de la fenêtre ne peut être modifiée par les différents éléments contenus dans celle-ci**, il faut donc que sa taille soit suffisante au départ.

```
  MyFrame *frame = new MyFrame(NULL,wxID_ANY,wxT(""), wxPoint(50,50),
wxSize(450,340));
  wxButton *button;
  wxLayoutConstraints *c;
  for(int i=1;i<=9;i++)
  {
    c=new wxLayoutConstraints; // création d'une contrainte
    button=new wxButton(frame,wxID_ANY,wxString::Format(wxT("%d"),i),
              wxDefaultPosition,wxDefaultSize,wxBU_TOP); // nouveau bouton
    c->centreX.SameAs(frame, wxCentreX); // centré en X
    c->centreY.SameAs(frame, wxCentreY); // centré en Y
    c->height.PercentOf(frame,wxHeight,100-10*i); // on réduit la taille de 10%
    c->width.PercentOf(frame,wxWidth,100-10*i);
    button->SetConstraints(c); // on fixe la contrainte
  }
  frame->SetAutoLayout(true); // on demande un calcul automatique
  frame->Show(true);
  SetTopWindow(frame);
  return true;
}
```

3. Utilisation des contrôleurs de taille : wxSizer

3.1 Introduction

Le principe est celui d'un placement dans des conteneurs graphiques avec un calcul de la taille de ceux-ci et une adaptation à la fois de la taille des conteneurs, mais aussi de la fenêtre. Cela permet de créer une fenêtre sans être obligé de spécifier une taille ou, si une taille est spécifiée, de voir celle-ci augmentée de façon automatique. Si vous changez de police sur votre système, la taille sera toujours correcte, y compris si vous re-compilez l'application sur une autre plate-forme. Vous utilisez la taille minimale du contrôle pour calculer la taille de la fenêtre ou de la zone allouée, avec cependant un petit souci pour les contrôles ne sachant pas ou ne pouvant pas calculer leur taille. Mais, le problème se pose aussi avec la méthode précédente.

3.2 la classe wxSizer : la base

Cette classe abstraite est la base du fonctionnement et donc il faudra utiliser une classe dérivée pour le placement. Il en existe principalement 5 pour une utilisation classique. La méthode est différente de la précédente pour le calcul des tailles : on indique une proportion et la largeur (ou la hauteur) totale est calculée sur la somme de toutes les proportions. Si vous mettez en largeur (ou hauteur suivant le sizer) 3 éléments ayant comme proportion 2, 1 et 1, alors le premier élément prendra 2/(2+1+1), soit 50% de la nouvelle taille alors que les deux autres prendront 25%. Cela fonctionne en augmentation ou diminution de taille avec la possibilité de fixer une taille minimale. Pour mieux comprendre, voici l'exemple précédent :

- un sélecteur de fichier à gauche de taille 150 pixels en largeur pouvant évoluer en hauteur ;
- à droite, 2 zones de texte pouvant évoluer en largeur et hauteur avec celle du haut en prenant 60% de la taille avec un espace de 10 pixels entre les deux zones.

La zone de texte du bas prendra toute la place de la zone contrairement au cas précédent, mais il ne pourra plus y avoir d'erreur de taille. Avec les sizers, on a forcément sur chaque zone un même nombre d'éléments, ce qui implique alors dans notre cas 2 sizers : un pour le sélecteur de fichier avec possibilité de changer la hauteur et un autre sizer pour les zones de texte avec possibilité de changer la largeur uniquement. Le deuxième sizer sera alors contenu dans le premier pour avoir aussi un déplacement possible en hauteur. On a alors le code suivant et le résultat figure 5. Le déplacement dans une seule dimension se fait à l'aide de l'élément `wxBoxSizer`.

```
wxFrame *frame = new wxFrame(NULL,wxID_ANY,wxT(""),
wxPoint(50,50));
wxTextCtrl *text_ctrl1 = new wxTextCtrl(frame, wxID_
ANY, wxT("Ceci est un texte d'exemple avant adaptation
automatique de la taille"),wxPoint(wxID_ANY, wxID_ANY),
wxSize(wxID_ANY, wxID_ANY),wxTE_MULTILINE);
```

```
wxTextCtrl *text_ctrl2 = new wxTextCtrl(frame, wxID_ANY,
wxT("texte2"),wxPoint(wxID_ANY, wxID_ANY), wxSize(wxID_ANY,
wxID_ANY),wxTE_MULTILINE);
// plus de taille ni de position
wxGenericDirCtrl *dir_ctrl= new wxGenericDirCtrl(frame,wxID_ANY);
// sizer avec expansion horizontale
wxBoxSizer *box1 = new wxBoxSizer(wxHORIZONTAL);
// idem en vertical
wxBoxSizer *box2 = new wxBoxSizer(wxVERTICAL);
// ici on bloque la taille et alignement de la bordure gauche,
droite, haut et bas
// avec une taille de 10 pixels
box1->Add(dir_ctrl,0,wxLEFT|wxRIGHT|wxTOP|wxBOTTOM|wxEXPAND,10);
// proportion de 6/10
box2->Add(text_ctrl1,6,wxRIGHT|wxTOP|wxEXPAND,10);
// un espace de 10 pixels de taille fixe
box2->Add(wxID_ANY,10,0,wxEXPAND,10);
// proportion 4/10
box2->Add(text_ctrl2,4,wxRIGHT|wxBOTTOM|wxEXPAND,10);
// le deuxième élément du sizer est le box2
box1->Add(box2,1,wxEXPAND);
// on fixe la taille minimale
box1->SetItemMinSize(dir_ctrl,wxSize(150,wxID_ANY));
 // ici 100 pixels
box1->SetItemMinSize(text_ctrl1,wxSize(100,wxID_ANY));
// la 10 pixels
box1->SetItemMinSize(text_ctrl2,wxSize(10,wxID_ANY));
// on permet au sizer de modifier la taille de la fenêtre
box1->SetSizeHints( frame );
frame->SetSizer(box1);
```

L'utilisation de la méthode `SetSizeHints( fenêtre )` permet de fixer de façon automatique la taille optimale pour la fenêtre et donc de ne plus avoir à y penser au moment de la création.

Le `wxBoxSizer` permet un agencement des contrôles sous forme de colonnes ou de lignes. Donc, suivant l'orientation, tous les éléments auront la même largeur ou hauteur. Dans notre exemple, `box1` fixe la même hauteur pour tous les éléments et `box2` fixe la même largeur pour les deux zones de texte.

3.3 La classe wxGridSizer

Ce contrôleur de taille permet de positionner en deux dimensions des éléments qui auront tous la même hauteur et la même largeur. Le plus grand impose la taille des autres. Cela permet de positionner un groupe de boutons ou de texte comme le montre la figure 6 avec le code associé dans le code suivant.

Figure 5 : Utilisation d'un wxBoxSizer

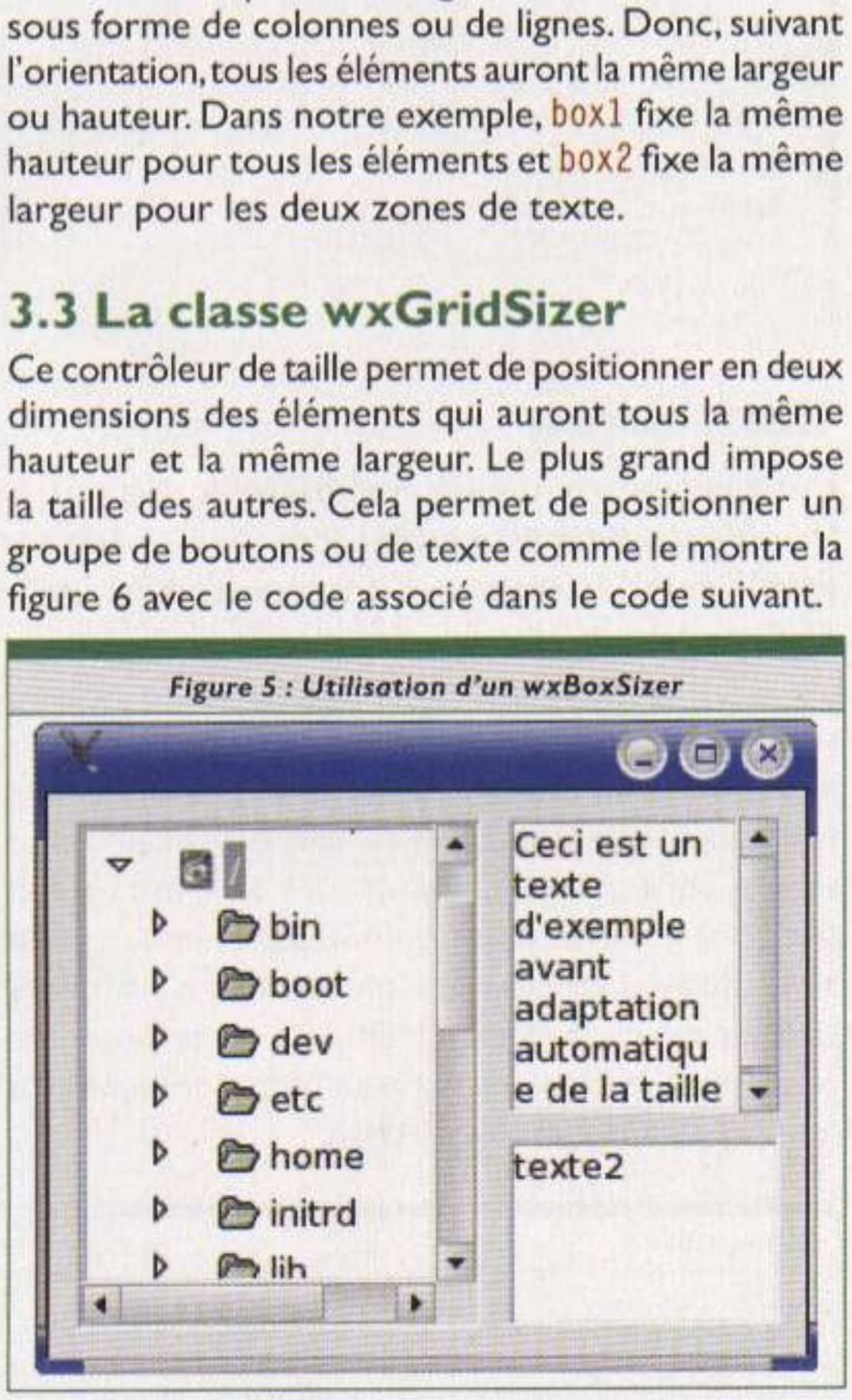


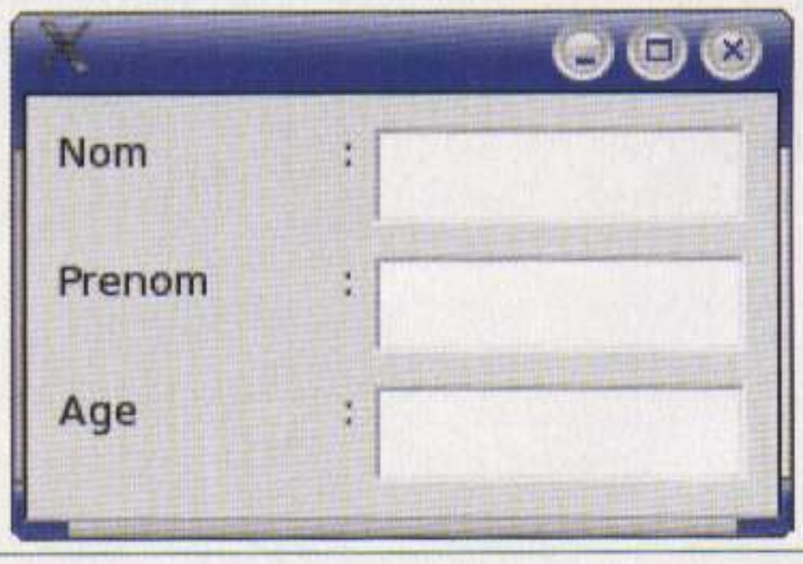
Figure 6 : Utilisation d'un wxGridSizer


```
// 3 lignes de 2 colonnes avec 10 pixels entre les
lignes et 5 entre les colonnes
  wxGridSizer *box=new wxGridSizer(3,2,10,5);
  for(int i=1;i<=3*2;i++) {
    wxButton *button=new wxButton(frame,wxID_ANY,wxString::
Format(wxT("%d"),i));
    box->Add(button,1,wxEXPAND);
  }
// pour modifier automatiquement la taille de la fenêtre
  box->SetSizeHints( frame );
// insertion du " sizer " dans la fenêtre
  frame->SetSizer(box);
```

3.4 La classe wxFlexGridSizer

Ce contrôleur de taille permet de positionner en deux dimensions des éléments dont les colonnes auront toutes la même largeur et les lignes la même hauteur. Le plus grand impose la taille des autres. Ce composant est idéal pour réaliser des boîtes de dialogue demandant des informations. Cela est illustré simplement par la figure 7 et le code suivant.

Figure 7 : Utilisation d'un wxFlexGridSizer



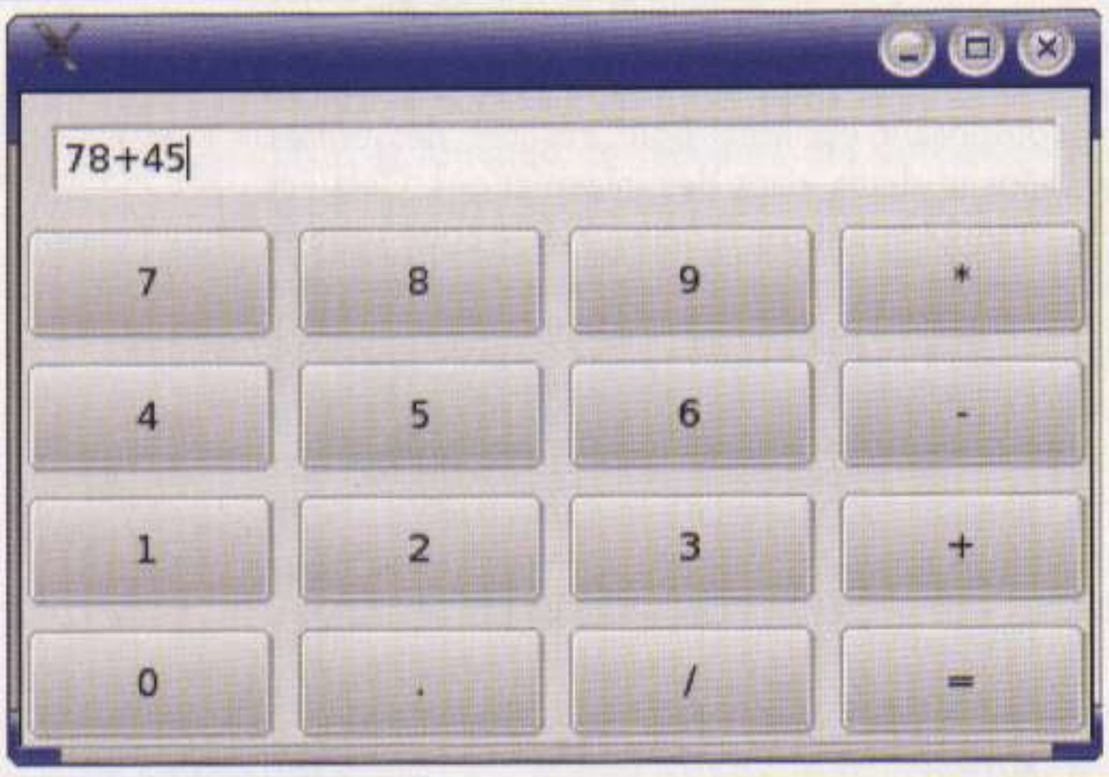

```
// 3x3 avec 10 pixels entre les colonnes et 5 pixels entre les lignes
wxFlexGridSizer *box=new wxFlexGridSizer(3,3,10,5);
box->Add(new wxStaticText(frame,wxID_ANY,wxT("Nom")),1,wxLEFT|wxTOP|wxEXPAND,10);
box->Add(new wxStaticText(frame,wxID_ANY,wxT(":")),1,wxTOP|wxEXPAND,10);
box->Add(new wxTextCtrl(frame,wxID_ANY),1,wxRIGHT|wxTOP|wxEXPAND,10);
box->Add(new wxStaticText(frame,wxID_ANY,wxT("Prenom")),1,wxLEFT|wxEXPAND,10);
box->Add(new wxStaticText(frame,wxID_ANY,wxT(":")),1,wxEXPAND);
box->Add(new wxTextCtrl(frame,wxID_ANY),1,wxRIGHT|wxEXPAND,10);
box->Add(new wxStaticText(frame,wxID_ANY,wxT("Age")),1,wxLEFT|wxBOTTOM|wxEXPAND,10);
box->Add(new wxStaticText(frame,wxID_ANY,wxT(":")),1,wxBOTTOM|wxEXPAND,10);
box->Add(new wxTextCtrl(frame,wxID_ANY),1,wxRIGHT|wxBOTTOM|wxEXPAND,10);

box->AddGrowableCol(0); // on indique  que la colonne 0 peut varier
box->AddGrowableCol(2); //
box->AddGrowableRow(0); // même chose pour les lignes 0 à 2
box->AddGrowableRow(1);
box->AddGrowableRow(2);
```

Bien sûr, il ne faut pas oublier d'indiquer que le sizer est dans la fenêtre et que la taille de la fenêtre est modifiée par le sizer. Par défaut, les colonnes et lignes ne changent pas de taille. Il faut spécifier cela en utilisant les méthodes `AddGrowableRow(ligne)` et `AddGrowableCol(colonne)`. Les numéros des lignes et colonnes **commencent par 0**. Dans cet exemple, les proportions sont de 0 qui ne signifie pas comme pour les exemples précédents une non-modification, mais, au contraire, que les éléments sont modifiés de façon proportionnelle. Mais, si on place une proportion sur un élément, il faut en mettre une sur tous les éléments variant et non laisser cela à 0.

3.5 Une calculatrice

Cet exemple permet juste de montrer la puissance de positionnement des sizers et de montrer une nouvelle méthode d'insertion de texte dans un `wxTextCtrl`. Il reste à faire le code complet de la calculatrice en récupérant le code de la touche [=]. La figure du programme est la figure 8 illustrant le code donné ci-après.

Figure 8 : Une calculatrice




```
class MyFrame : public wxFrame
{
  public:
// constructeur
   MyFrame(wxWindow *parent,wxWindowID id);
// traitement des clics bouton
   void OnButton(wxCommandEvent& event);
   wxTextCtrl* GetAffichage();
  private:
// le texte de l'affichage
   wxTextCtrl *affichage;
   DECLARE_EVENT_TABLE()
};
MyFrame::MyFrame(wxWindow *parent,wxWindowID id) :
        wxFrame(parent,id,wxEmptyString),affichage(new
wxTextCtrl(this,wxID_ANY))
{ // rien à faire car l'affichage est déjà créé
}
```

```
void MyFrame::OnButton(wxCommandEvent& event)
{
// on récupère le bouton et son label est inséré dans
l'affichage
  wxButton* button;
  button=dynamic_cast<wxButton*>(event.GetEventObject());
  if (button!=NULL) affichage->WriteText(button-
>GetLabel());
}

bool MyApp::OnInit()
{
MyFrame *frame = new MyFrame(NULL,wxID_ANY);

  // les touches
  wxString texte[]={wxT("7"),wxT("8"),wxT("9"),wxT(
"*"),wxT("4"),wxT("5"),wxT("6"),wxT("-"),wxT("1"),w
xT("2"),wxT("3"),wxT("+"),wxT("0"), wxT("."),wxT("/
"),wxT("=")};
// extension en vertical
  wxBoxSizer *box=new wxBoxSizer(wxVERTICAL);
  box->Add(frame->GetAffichage(),1,wxLEFT|wxRIGHT|wxTO
P|wxBOTTOM|wxEXPAND,10);
// 16 touches avec 5 pixels d'espace
  wxGridSizer *grid=new wxGridSizer(4,4,5,5);
  for(int i=0;i<16;i++)
// création des boutons
    grid->Add(new wxButton(frame,wxID_ANY,texte[i]),1
,wxEXPAND);
 // on récupère 4/5 de la place
  box->Add(grid,4,wxEXPAND);
// le sizer commande la taille de la fenêtre
  box->SetSizeHints( frame );
// on insère le sizer
  frame->SetSizer(box);
  frame->Show(true);
  SetTopWindow(frame);
  return true; // tout va bien
}
```

Conclusion

Cette seconde partie sur les contrôles présente la façon de procéder pour positionner des éléments sur une fenêtre (ou un panneau) sans se soucier de la taille de l'écran ou de la police des caractères. C'est une méthode « automatique » permettant une utilisation simple des logiciels de création d'interfaces. Les deux méthodes sont assez simples, mais il paraît évident maintenant que la seconde est plus sûre et aussi plus simple. Sur le dernier exemple, il y a 16 boutons et une zone de texte, soit 17x4=68 contraintes à gérer, alors qu'avec la première méthode il n'y a que 2 sizers dans cette partie. Même en automatisant le processus, on se retrouve avec un programme plus lourd à comprendre et à écrire.

Olivier Corrio,

olivier.corrio@gmail.com

Maurice Szmurlo

Développement de services SMS

Avec l'avènement de la téléphonie mobile, on constate un rapprochement très fort entre les technologies télécom et l'informatique. Là où il y a encore une quinzaine d'années un service devait être implémenté sur l'ensemble des commutateurs (on utilise généralement le terme anglais « switch ») télécom d'un opérateur, il sera aujourd'hui déporté sur des serveurs informatiques. Ce rapprochement est d'autant plus visible sur les services dits « à valeur ajoutée » pour lesquels les plateformes ont une patte connectée sur le réseau télécom via les protocoles de la pile SS7 (voir [1] ou [2] pour une introduction au « Signaling System 7 ») et une autre sur le réseau IP, permettant, d'une part, le développement et le déploiement rapides d'applications et, d'autre part, leur interconnexion avec le système d'information de l'opérateur pour la facturation, pour la gestion des abonnements, etc. Parmi ce type de services, on peut citer le téléchargement de logos et de sonneries, les musiques qui remplacent les sonneries d'attente ou les services à base de SMS tels le chat, les flashs d'alerte divers et variés (actualités, bourse, banque...), etc. C'est sur le développement de services à base de SMS que nous allons nous pencher au cours de cet article.

Travailler avec les SMS peut, à première vue, paraître un peu passéiste lorsque la tendance se porte sur les MMS, les messageries instantanées de type Messenger ou MSN, issues du monde Internet, voire le mail. L'augmentation du trafic SMS chez tous les opérateurs dément cette affirmation. Le SMS est en effet un moyen très simple de communication, disponible pour tous les abonnés d'un opérateur sans qu'il soit nécessaire de définir (ni de mémoriser) des identifiants spécifiques, style adresses mail ou pseudos, qu'il est possible de recevoir sur tous les terminaux du marché, même les plus simples. Développer des services SMS peut donc être très intéressant pour les opérateurs eux-mêmes, pour des fournisseurs de services tiers ou pour des sociétés telles des banques pour fournir des services de type alertes.

Dans la suite de cet article, nous allons préciser le fonctionnement technique de la transmission d'un SMS, discuter brièvement du protocole utilisé, SMPP et d'une bibliothèque Java, `smppapi`, qui implémente ce protocole. Nous mettrons en pratique ces discussions en implémentant deux services, l'un très simple, permettant d'envoyer un SMS en ligne de commande, l'autre un peu plus complexe faisant du *gatewaing mail*. Trois programmes de démonstration sont fournis : les deux services et un squelette d'application permettant de réaliser des expériences sur `smppapi`. Ils sont situés dans le répertoire `dev`. Ces répertoires ont tous la même structure : le répertoire `dev/<programme>` contient un `.jar` exécutable (à lancer par la commande `java -jar <programme>.jar`) ainsi qu'un fichier `build.xml` pour la compilation par Ant. `dev/<programme>/src/etc` contient les fichiers de configuration et `dev/<programme>/src/org/buguigny/<programme>` contient les sources java.

Les SMS, comment ça marche ?

Même si, pour développer des applications SMS, il n'est pas nécessaire de connaître l'architecture détaillée de l'infrastructure technique d'un opérateur mobile, il peut néanmoins être intéressant d'en avoir une idée générale.

Éléments techniques d'un réseau GSM

Un opérateur GSM se base sur un certain nombre d'éléments techniques :

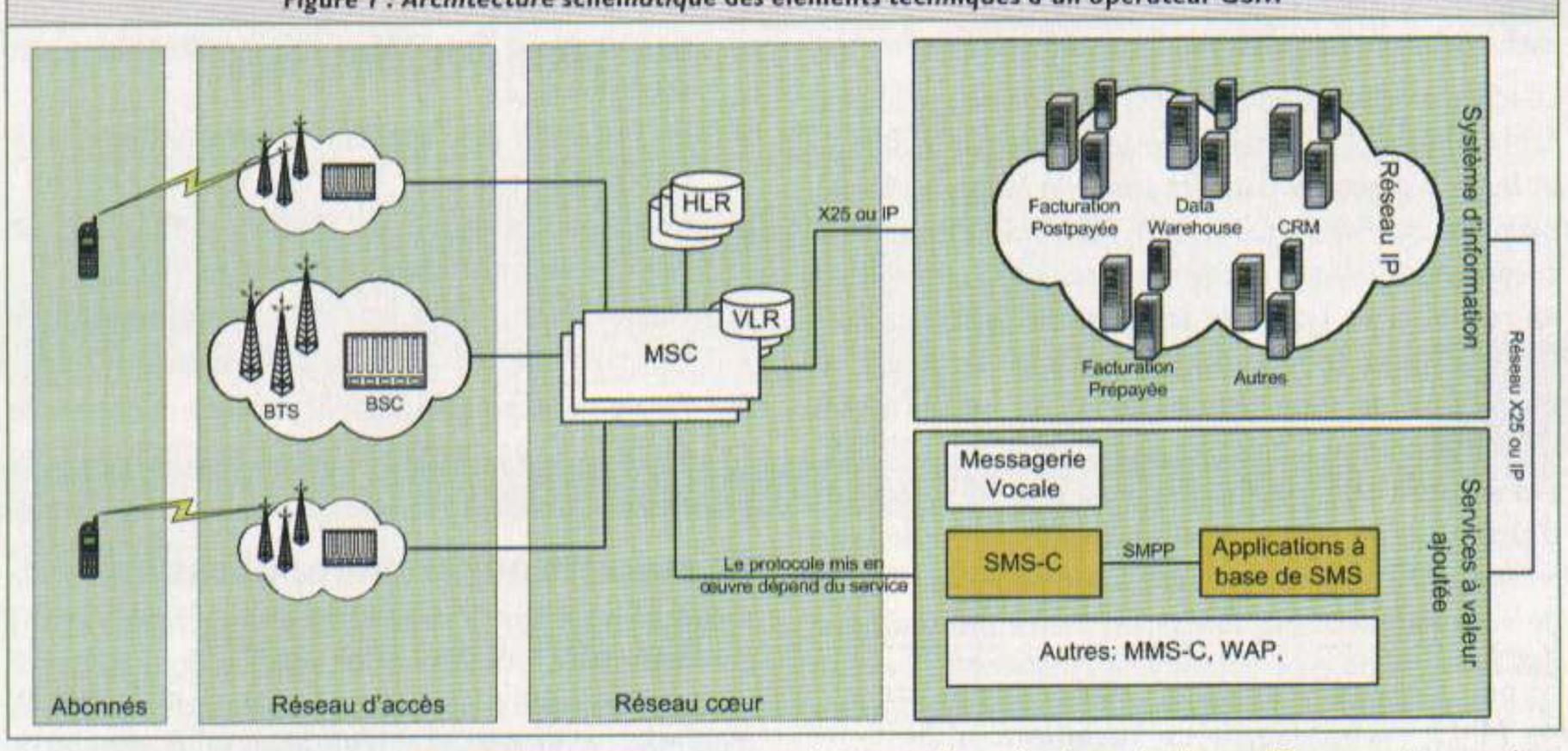
- Le premier élément est le terminal de l'abonné, appelé « MS » (*Mobile station*) dans le vocabulaire GSM. Pour ce qui nous concerne, la MS peut envoyer ou recevoir des SMS.
- Le réseau d'accès est composé d'antennes (BTS, *Base Tranceiver Station*) et de contrôleurs d'antennes (BSC, *Base Station Controler*) qui permettent aux terminaux de se connecter (d'accéder) au réseau par radio.
- Le réseau cœur est connecté au réseau d'accès et reçoit les demandes de l'utilisateur. Il est composé d'un ou plusieurs MSC (*Mobile Switching Center*, le commutateur ou *switch*), chacun couvrant une zone géographique et de bases de données techniques. Le MSC vérifie si l'abonné appelant a les autorisations nécessaires pour le service demandé en interrogeant des éléments tiers, puis assure l'itinérance lorsqu'un abonné appelle en se déplaçant, l'acheminement des appels, ou délègue la réalisation du service demandé à une plate-forme tiers (par exemple le SMS-C dans le cas d'envoi de SMS). Les bases techniques sont au nombre de trois : le HLR (*Home Location Register*), le VLR (*Visitor Location Register*) et l'AUC (*AUthentication Center*). Le HLR liste les services auxquels l'abonné a accès (par exemple le *roaming*, les appels internationaux, la réception ou l'émission

d'appels, etc.). Le VLR est, d'une part, une copie de certaines informations contenues dans le HLR, mais il contient également les informations de localisation de l'abonné qui permettent au MSC de réaliser l'acheminement des appels via le bon BSC sur le bon BTS. Finalement, l'AUC gère les clefs secrètes utilisées pour authentifier les demandes de service et pour chiffrer les communications. Les AUC sont généralement associés aux HLR. Notons que dans le cas de connexions data GPRS ou EDGE, des éléments additionnels doivent être mis en place (SGSN et GGSN).

- En aval du réseau cœur, on trouve deux ensembles d'éléments : le système d'information et les plateformes de service. Le système d'information est en charge de la gestion commerciale du client. Typiquement, il contient des applications de type CRM, de facturation ou de *data warehouse*. Les plateformes de service permettent de rendre des services supplémentaires aux abonnés comme par exemple la messagerie vocale, l'envoi de MMS ou l'envoi de SMS.

La figure 1 présente de manière schématique et simplifiée les éléments discutés ci-dessus. Les lecteurs n'ayant pas peur d'entrer dans le vif du sujet peuvent se référer à **[3]**.

Figure 1 : Architecture schématique des éléments techniques d'un opérateur GSM



Fonctionnement des SMS

Il y a trois classes de SMS :

- Les messages venant d'un mobile : l'abonné envoie un message. On parle alors de SMS-MO, MO pour *Mobile Originating*.
- Les messages arrivant à un mobile : l'abonné reçoit un message. On parle alors de SMS-MT, MT pour *Mobile Terminating*.
- Les messages en diffusion (*broadcast*) : l'opérateur envoie le même message à un ensemble d'abonnés localisés dans une zone géographique.

Le service SMS requiert une plate-forme dédiée appelée « SMS-C » (*SMS Center*). Le SMS-C (en orange sur la figure 1) est connecté au switch en utilisant les protocoles de la pile SS7. Il est identifié sur le réseau télécom par un numéro de téléphone afin que le MSC puisse lui transmettre les messages à envoyer. Le SMS-C est connecté au système d'information, en général par un lien TCP/IP qui permet, par exemple, de recueillir les informations nécessaires à la facturation du client. Finalement, le SMS-C peut également être connecté à des applications pouvant recevoir ou envoyer des SMS, les services SMS. Le protocole utilisé dans ce cas est SMPP (*Short Message Peer to Peer*) dont nous parlerons plus tard. Ce lien peut être utilisé, par exemple, par la messagerie vocale pour prévenir un abonné qu'un message a été déposé dans sa boîte.

Lors de l'envoi d'un SMS par un client (SMS-MO), le terminal code le corps du message et l'envoie au MSC avec deux informations supplémentaires : le numéro du destinataire et le numéro du SMS-C à utiliser. Ce dernier est configuré dans le téléphone et permet au MSC de relayer le SMS au SMS-C qui le stocke. Si le message n'a pu être délivré au SMS-C, le réseau envoie un acquittement négatif. Le SMS-C tente ensuite d'envoyer le SMS vers le destinataire (SMS-MT) à travers le MSC. Si la première tentative échoue (par exemple, si le terminal du destinataire est éteint), le message est stocké et d'autres tentatives sont réalisées à intervalles réguliers. Au bout d'un certain nombre d'échecs, ou après un certain délai, le message est effacé.

Le destinataire d'un message peut être un abonné de l'opérateur ou celui d'un autre opérateur, mais peut également être une application, typiquement identifiée par un numéro court (mais ce n'est pas obligatoire), disons le 100. Dans ce cas, le SMS-C doit être configuré de sorte que si le numéro du destinataire est le 100, il ne tente pas de l'envoyer vers le MSC, mais utilise le lien SMPP pour déclencher l'application destinataire. De la même manière, l'expéditeur peut aussi être une application : pour envoyer un message à un abonné, elle doit se connecter sur le SMS-C, créer un paquet SMPP avec au minimum le numéro du destinataire et le corps du message et l'envoyer au SMS-C. Dans le jargon SMPP, les applications et

les services sont nommés « ESME » pour *External Short Message Entity*.

Le protocole SMPP : Short Message Peer to Peer

SMPP est un protocole ouvert défini par le SMS Forum (**www.smsforum.net**). La version la plus récente est la 5.0, mais la plupart des SMS-C implémentent la version 3.4, voire la 3.3. Les spécifications pour les versions 3.4 et 5.0 sont disponibles sur le site du SMS Forum.

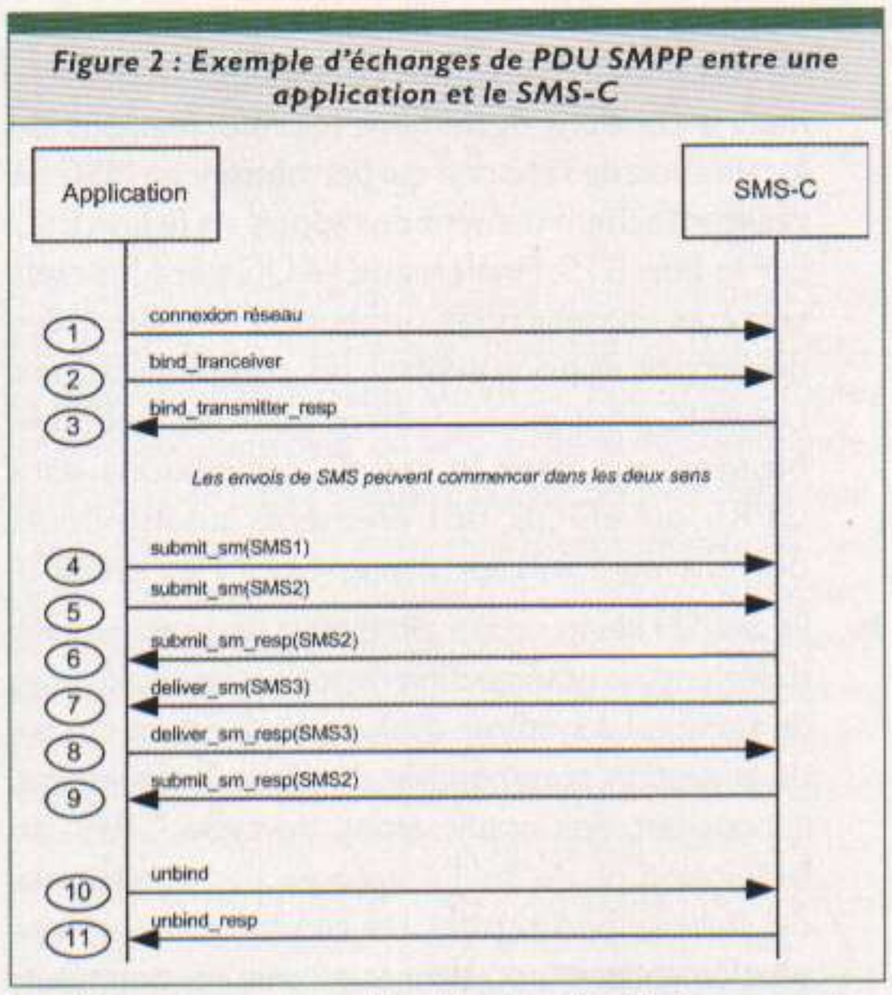
D'un point de vue technique, SMPP est un protocole binaire de niveau 7 (couche application) et s'appuie sur un transport basé sur X25 ou TCP/IP. C'est un protocole où l'initiative d'un échange peut être prise soit au niveau du SMS-C (envoi d'un SMS vers le service qui se traduit par l'envoi d'une requête SMPP depuis le SMS-C vers ce service) ou au niveau du service (le service envoie une requête SMPP vers le SMS-C). SMPP est également asynchrone : le service peut envoyer (ou recevoir) plusieurs SMS avant de recevoir (envoyer) les acquittements. Les deux entités, SMS-C et service, doivent donc agir comme des serveurs. L'implémentation directe par une application d'une couche SMPP peut par conséquent s'avérer assez complexe. La bibliothèque Java que nous allons utiliser cache autant que possible cette difficulté.

Le schéma de fonctionnement typique d'une application est le suivant : l'application se connecte sur le SMS-C et les échanges de paquets (dans le langage réseau, on parle de *Packet Data Unit*, PDU) SMPP peuvent commencer dans un sens comme dans l'autre. Il y a en réalité trois types de connexions possibles qu'il faut choisir en fonction des besoins du service. Une connexion de type *transmitter* (TX) ne permet à l'application que d'envoyer des messages au SMS-C. On utilisera ce type de connexion pour des services d'alerte ou de notification. Une connexion de type *receiver* (RX) ne donnera à l'application que la possibilité de recevoir des SMS. Elle pourra être utilisée pour des services de type *gatewaying*. Finalement, il existe un mode mixte, *transceiver* (TRX), où l'application peut envoyer et recevoir des SMS.

SMPP est composé d'un peu moins d'une trentaine d'opérations et à chacune d'entre elles est associé un PDU. Ceux-ci sont composés d'un en-tête suivi d'un corps dont le contenu dépend du type de l'opération. Les opérations implémentées par les PDU sont de type connexion, déconnexion, envoi de SMS, réception de SMS, envoi en masse, acquittements, etc. A chaque requête doit correspondre un acquittement. Nous n'entrerons pas plus dans les détails des PDU SMPP : la bibliothèque que nous utiliserons définit des objets pour chacun qu'entre eux et gère leur encodage. La norme complète est disponible sur **[4]**

Pour terminer cette discussion sur SMPP, analysons un échange de PDU entre une application et un SMS-C. Cet échange est illustré figure 2.

Figure 2 : Exemple d'échanges de PDU SMPP entre une application et le SMS-C



- Pour commencer, l'application établit la connexion réseau sur le SMS-C en (1). Ceci n'est pas une opération SMPP, mais tout simplement l'ouverture d'une *socket* vers un serveur.
- L'application s'enregistre sur le SMS-C en tant que transceiver lui permettant d'envoyer et de recevoir des SMS (2). L'acquittement est envoyé en (3).
- En (4) et en (5) l'application envoie deux SMS. L'acquittement pour le SMS 2 est reçu en (6).
- L'application reçoit un SMS du SMS-C (7) et renvoie aussitôt l'acquittement en (8).
- L'application reçoit en (9) l'acquittement pour l'envoi du SMS 2 qui a pu être retardé.
- Finalement, en (10), l'application se déconnecte du SMS-C et reçoit en (11) l'acquittement de cette demande.

Faire fonctionner une application SMS

Avant de nous lancer dans le développement, voyons comment faire fonctionner un service. Pour résumer ce qui précède, nous avons besoin : d'un SMS-C, d'un MSC, d'un VLR, d'un HLR, d'un AUC, d'un BCS, d'une BTS et sans doute de quelques éléments supplémentaires que j'aurais oubliés... Heureusement, les choses sont plus simples. La première solution est de négocier (commercialement), avec un opérateur qui possède toute l'infrastructure, la mise à disposition d'un compte (lien réseau vers son SMS-C et droits d'accès). Les opérateurs télécom ne sont intéressés par ce type d'opération que si le trafic SMS engendré est important : plusieurs dizaines voire centaines de milliers de SMS par jour. La deuxième solution est de se rapprocher d'un *broker* de SMS qui, lui, aura négocié avec des opérateurs télécom des volumes importants et qui vous revendra l'envoi ou la réception de SMS à la dizaine, à la centaine, au millier, etc. On en trouve de très nombreux en effectuant une recherche sur le web. La plupart de ces brokers ne permettent

malheureusement que l'envoi et offrent des interfaces propriétaires souvent basées sur HTTP (HTTP pur, XML sur HTTP, etc.). Pour un fonctionnement plus « artisanal » (rien de péjoratif dans cet adjectif), on peut utiliser Kannel **[5]** qui peut agir en tant que SMS-C une fois couplé à un modem GSM, mais cette direction n'a pas été creusée.

Développement de services SMS

Les outils nécessaires au développement

Même si vous avez accès à un SMS-C, aucun opérateur ne vous laissera faire la mise au point de votre application directement sur sa plate-forme. Par conséquent, il est nécessaire d'utiliser un simulateur qui permettra en plus de faire des tests plus poussés que sur un SMS-C en production. En recherchant sur le WEB, on trouve facilement deux simulateurs : celui de LogicaCMG **[6]**, sous licence Logica Open Source License v1.0, et « SMPPSim » de Selenium Software **[7]**, sous licence GLP v2. Pour nos expérimentations, nous utiliserons ce dernier. Outre la simulation d'un SMS-C, SMPPSim permet également de simuler l'envoi de SMS à partir d'un terminal. Ceci est réalisé par une interface Web.

L'installation de SMPPSim est très facile. Téléchargez `SMPPSim.tar.gz` et décompressez-le dans un répertoire. La racine contient le fichier `startsmppsim.sh` qui permet de lancer le simulateur. Le répertoire `conf` contient le fichier `smppsim.props` qui permet de configurer le simulateur. Il y a de nombreuses options, les principales étant :

```
# Le port d'écoute SMPP
SMPP_PORT=2775

# login et mot de passe pour les accès SMPP
SYSTEM_ID=smppclient
PASSWORD=password

# Le port d'écoute du serveur web pour
l'administration et la simulation
# d'envoi à partir d'un terminal
HTTP_PORT=88
```

Comme expliqué dans le paragraphe consacré à SMPP, il est préférable d'utiliser une API capable de gérer les aspects de codage et de décodage des PDU, mais également l'asynchronisme inhérent au protocole. Là aussi, une rapide recherche donne deux résultats : la « Java SMPP Library de Logica » **[6]** et « smppapi » que nous utiliserons. `smppapi` est disponible sur Sourceforge en LGPL (voir **[8]**).

Qu'il s'agisse de SMPPSim ou `smppapi`, les deux fonctionnent sous Java. La dernière version de SMMPSim requiert Java 1.5. `smppapi` utilise Log4J et Commons-Logging que nous inclurons dans les jars exécutables ou dans le `CLASSPATH`. Finalement, Ant nous aidera dans les compilations.

Prise en main de smppapi

La meilleure manière de prendre en main une API est de l'utiliser. Nous commencerons donc par un exemple très simple : un programme en ligne de commande qui enverra un message à un abonné. Bien entendu, puisque le SMS-C est un simulateur, aucun SMS réel ne partira. Notre programme, `smssender`, sera appelé de la manière suivante : `java -jar smssender.jar 0123456789 Un petit message`.

Le programme sera divisé en deux fichiers. `SMSSender.java` définit la classe `SMSSender`. Le constructeur positionne les paramètres de connexion. `open()` ouvrira la connexion, `close()` la fermera. Finalement, `sendMessage()` enverra le message au destinataire. Le code dans `Main.java` analyse la ligne de commande, crée un objet `SMSSender` et appelle les méthodes pour réaliser l'envoi.

Le source complet de `SMSSender` est présenté ci-dessous. Nous l'analyserons morceau par morceau dans les paragraphes suivants.

```
01:package org.buguigny.smssender;
02:
06:import ie.omk.smpp.message.SubmitSM;
07:import ie.omk.smpp.Connection;
08:import ie.omk.smpp.message.BindResp;
09:import ie.omk.smpp.message.SMPPPacket;
10:import ie.omk.smpp.Address;
11:import ie.omk.smpp.message.SMPPResponse;
14:import ie.omk.smpp.message.UnbindResp;
16:import org.apache.commons.logging.Log;
17:import org.apache.commons.logging.LogFactory;
18:
19:public class SMSSender {
20:
21:  // l'objet Connection qui va gérer les transactions réseau
22:  private Connection cnx;
23:  // Le nom du serveur qui héberge le SMS-C
24:  private String host;
25:  // Le port TCP/IP sur lequel le SMS-C écoute les connexions SMPP
26:  private int port;
27:  // Le login et le mot de passe pour se connecter au SMS-C
28:  private String uname;
29:  private String passwd;
30:  // Un logguer pour des traces propres
31:  private static Log log = LogFactory.getLog(SMSSender.class);
32:
33:  // constructeur passif: initialisation des membres
34:  public SMSSender(String host, int port, String uname, String passwd) {
35:    cnx = null;
36:    this.host = host;
37:    this.port = port;
38:    this.uname = uname;
39:    this.passwd = passwd;
40:  }
41:
42:  // Connexion au SMS-C
43:  public boolean open() {
44:    try {
45:      cnx = new Connection(host, port, false);
46:      BindResp resp = cnx.bind(Connection.TRANSMITTER, uname, passwd, "SMSSender");
47:      if(resp.getCommandStatus() != 0) return false;
48:    }
49:    catch(Exception e) {
50:      log.fatal("Problème lors du open(): "+e);
51:      return false;
52:    }
53:    return true;
54:  }
55:  // Déconnexion du SMS-C
56:  public void close() {
```

```
57:     try {
58:       UnbindResp resp = cnx.unbind();
59:     }
60:     catch(Exception e) {
61:       log.error("Problème dans close(): "+e);
62:     }
63:   }
64:
65:   // Envoi de message
66:   public boolean sendMessage(String srcNO, String dstNO, String mess) {
67:     SubmitSM sm = null;
68:     try {
69:       sm = (SubmitSM)cnx.newInstance(SMPPPacket.SUBMIT_SM);
70:     }
71:     catch(Exception e) {
72:       log.fatal("Problème à la création de SubmitSM: "+e);
73:       return false;
74:     }
75:
76:     Address dst = new Address(0, 0, dstNO);
77:     Address src = new Address(0, 0, srcNO);
78:     sm.setDestination(dst);
79:     sm.setSource(src);
80:     sm.setMessageText(mess);
81:     SMPPResponse resp = null;
82:     try {
83:       resp = cnx.sendRequest(sm);
84:     }
85:     catch(Exception e) {
86:       log.error("Problème à l'envoi du message: "+e);
87:       return false;
88:     }
89:
90:     if(resp.getCommandStatus() != 0) {
91:       log.error("Ne peut pas envoyer le message.");
92:       return false;
93:     }
94:     return true;
95:   }
96:}
```

Connexion au SMS-C: méthode open()

La méthode `open()` est définie à partir de la ligne 42. La première opération (ligne 45) met en place une connexion au niveau réseau sur le SMS-C. Les deux premiers paramètres du constructeur de l'objet `Connection` sont le nom ou l'adresse IP du SMS-C (`localhost` pour nous) et le port. Le port doit avoir la valeur que nous avons fixée dans le fichier de paramètres, 2775. Le troisième paramètre indique à l'API si elle doit traiter les messages de manière synchrone (`false`) ou asynchrone (`true`). Puisque notre application n'envoie qu'un seul SMS, la connexion sera synchrone. Le second point est d'enregistrer l'application auprès du SMS-C en précisant dans quel sens les SMS vont être échangés (`TRANSMITTER` dans le sens service vers SMS-C), les accréditations (*login* et mot de passe comme indiqués dans le fichier de configuration) et finalement le nom du service qui se connecte. La création de la `Connection` ou l'appel à `bind()` peuvent lancer des exceptions : mauvais nom de *host*, mauvais mot de passe, etc. Pour alléger le source, toutes ces exceptions sont attrapées par un `catch(Exception e)` global. L'appel à `bind()` envoie un PDU au SMS-C qui renvoie un acquittement sous forme de PDU représenté par un objet de la classe `BindResp`. La méthode `getCommandStatus()` renvoie un entier non nul si une erreur est survenue et zéro sinon. Notons finalement qu'un `bind()` résulte en une négociation entre l'application et le SMS-C sur la version du protocole à utiliser, sur les éventuels paramètres optionnels acceptés, etc. Le résultat de cette négociation est mémorisé par l'objet `Connection` et servira à la gestion d'erreurs lors de la création de PDU, comme nous le verrons plus loin.

Si on lance le programme et si on regarde les traces côté simulateur, on voit, entre autres, les lignes suivantes (pour plus de lisibilité, celles-ci ont été tronquées de la date ainsi que de la criticité ; des numéros de lignes ont par contre été rajoutés, de même que dans le code au-dessus) :

```
01: : BIND_TRANSMITTER:
02: Hex dump (50) bytes:
03: 00000032:00000002:00000000:00000001:
04: 736D7070:636C6965:6E740070:61737377:
05: 6F726400:534D5353:656E6465:72003400:
06: 0000
07: cmd_len=50,cmd_id=2,cmd_status=0,seq_no=1,system_id=smppclient
08: password=password,system_type=SMSSender,interface_version=52,addr_ton=0
09: addr_npi=0,address_range=
10:
11: New transmitter session bound to SMPPSim
12: : BIND_TRANSMITTER_RESP:
13: Hex dump (24) bytes:
14: 00000018:80000002:00000000:00000001:
15: 534D5050:53696D00:
16: cmd_len=0,cmd_id=-2147483646,cmd_status=0,seq_no=1,system_
id=SMPPSim
```

La ligne 1 affiche le nom du PDU reçu et les lignes suivantes le *dump* hexadécimal (3 à 6) et les champs (lignes 7 à 9) de ce PDU. Nous n'entrerons pas dans le détail des composants du PDU. Comme indiqué précédemment, toute requête doit entraîner une réponse dont le PDU est dumpé dans les lignes 12 à 16.

Déconnexion du SMS-C : méthode close()

La déconnexion du SMS-C est réalisée très simplement par l'appel de la méthode `cnx.unbind();` où `cnx` est la `Connection` initialisée dans la méthode `open`.

Envoi d'un SMS: méthode sendMessage()

La méthode `sendMessage` prend trois paramètres. `srcNO` est le numéro de l'abonné qui envoie le message et qui sera transmis tel quel au destinataire. Dans notre cas, il s'agit de l'application. La validité de ce numéro n'est pas vérifiée si bien qu'il est même possible d'y mettre une chaîne de caractères. `dstNO` est le numéro de téléphone du destinataire. Pour que le SMS soit acheminé correctement, il est évident que ce numéro doit d'une part être correct et attribué à un abonné valide. Finalement, `mess` est le message à transmettre. L'envoi de message est réalisé en trois étapes : création du PDU d'envoi, initialisation de certains membres du PDU, puis finalement envoi avec vérification de l'acquittement. Les messages doivent avoir une longueur de 160 caractères au maximum et sont encodés sur 7 bits, ce qui interdit l'utilisation

d'accents. D'autres méthodes d'encodage sont possibles (8 ou 16 bits), au prix de la longueur du message. La création de tous les PDU doit être réalisée par la méthode `newInstance` de la classe `Connection` qui initialisera, en fonction de la connexion ouverte, le numéro de séquence du paquet, ainsi que la version de SMPP utilisée (`smppapi` sait gérer SMPP 3.3 ou 3.4). `newInstance` prend en paramètre un entier qui identifie le type de PDU à créer. Les entiers à passer en paramètre sont à choisir parmi les constantes définies à cet effet dans la classe `SMPPPacket`. Ainsi `sm = (SubmitSM)cnx.newInstance(SMPPPacket.SUBMIT_SM);` crée un PDU de type `SUMBIT_SM`. `newInstance` peut lancer deux types d'exception : `BadCommandIDException` si l'entier passé à `newInstance` ne correspond à aucun type de PDU et `VersionException` si le PDU demandé n'est pas défini par la version de SMPP gérée par cette connexion.

Les membres du PDU `SUMBIT_SM` à initialiser sont les numéros de l'expéditeur, celui du destinataire et le message à proprement parler. Les numéros de téléphone sont gérés pas les objets de la classe `Address`. La création d'un objet `Address` requiert trois informations : le *Type Of Number* (ou TON), le *Numbering Plan Indicator* (ou NPI), et enfin le numéro du destinataire à proprement parler. Le TON et le NPI sont définis dans des normes internationales (voir **[9]**, page 81, la norme Q931 de L'Union Internationale des Télécommunications, ITU) et ont pour objectif d'aider le MSC au routage du message. Le TON spécifie si le numéro est un numéro national, international, court, etc. et peut prendre les valeurs suivantes : 0, 1, 2, 3, 4, 6 et 7. « 0 » représente la valeur « inconnu ». Le NPI spécifie le type de plan de numérotation, par exemple, appel voix, appel télex, etc. et prend les valeurs suivantes : 0, 1, 3, 4, 8, 9, 15. Comme précédemment, « 0 » représente la valeur « inconnu ». Ces constantes ne sont pas définies dans la bibliothèque (et donc non vérifiées), si bien que la valeur qui sera indiquée dans le constructeur sera insérée dans le PDU et transmise au SMS-C, puis au MSC avec de possibles conséquences négatives sur le routage au pire, ou un acquittement négatif au mieux. Nous utiliserons « 0 », c'est-à-dire « inconnu » pour ces deux valeurs. Le troisième paramètre du constructeur est le numéro de téléphone. L'initialisation du PDU `SUBMIT_SM` est réalisée lignes 76 à 80.

L'envoi du message se fait lignes 81 à 93. La méthode `sendRequest` de la classe `Connection` envoie les PDU de tout type, lance des exceptions en cas de problème et renvoie un acquittement en mode synchrone ou `null` en mode asynchrone (nous y reviendrons).

Analyse des traces côté SMS-C

Les traces correspondant à l'envoi du message avec la ligne de commande `java -jar smssender.jar 0612345678 Ceci est un sms` sont présentées ci-après. On observe sur le dump (lignes 7 à 12) que de nombreux champs additionnels ont été initialisés à des valeurs par défaut. Le dernier, `short_message` contient le message que nous souhaitons envoyer.

```
01: INFO    : Standard SUBMIT_SM:
02: INFO    Hex dump (61) bytes:
03: INFO    0000003D:00000004:00000000:00000002:
04: INFO    00000032:35310000:00303631:32333435:
05: INFO    36373800:00000000:00000000:000F4365:
06: INFO    63692065:73742075:6E20736D:73
07: INFO   cmd_len=61,cmd_id=4,cmd_status=0,seq_no=2,service_type=,source_addr_ton=0
08: INFO    source_addr_npi=0,source_addr=251,dest_addr_ton=0,dest_addr_npi=0
09: INFO    dest_addr=0612345678,esm_class=0,protocol_ID=0,priority_flag=0
10: INFO    schedule_delivery_time=,validity_period=,registered_delivery_flag=0
11: INFO   replace_if_present_flag=0,data_coding=0,sm_default_msg_id=0,sm_length=15
12: INFO    short_message=Ceci est un sms
13: INFO
14: WARNING Validity period is not set: defaulting to 5 minutes from now
15: INFO    Generated default validity period=061116110122000+
```

Mode asynchrone

Nous l'avons dit précédemment, SMPP est par nature asynchrone. L'asynchronisme permet d'améliorer les performances de l'application, mais complique le code. La bibliothèque `smppapi` nous fournit les outils pour le gérer.

Fonctionnement de l'API en mode asynchrone

En mode asynchrone, l'application doit pouvoir répondre aux évènements au fur et à mesure qu'ils sont reçus. Lors de la création d'une `Connection`, l'API crée un *thread* qui effectuera les opérations d'échange de PDU avec le SMS-C. Lors de la réception d'un PDU, ce thread doit fournir le PDU à l'application. Une façon élégante de procéder est d'utiliser le *pattern* Observer et d'implémenter dans l'application l'interface `ConnectionObserver`. Celle-ci définit deux méthodes qui doivent être implémentées dans le service et qui sont appelées par l'API :

- `public void update(Connection cnx, SMPPEvent evt)` est appelée lorsque l'API souhaite communiquer au service des informations sur l'état du thread d'écoute. La méthode `int getType()` de la classe `SMPPEvent` renvoie l'une des trois constantes suivantes : `RECEIVER_EXCEPTION`, `RECEIVER_EXIT` ou `RECEIVER_START`, respectivement lorsque le thread d'écoute reçoit une exception non fatale, c'est-à-dire lorsque l'exception ne le fait pas se terminer, lorsqu'il se termine normalement ou du fait d'une exception fatale ou lorsqu'il démarre.
- `public void packetReceived(Connection cnx, SMPPPacket evt)` est appelée lorsqu'un paquet est reçu sur la connexion réseau. La classe `SMPPPacket` définit des constantes pour l'ensemble des PDU susceptibles d'être reçus (par exemple `DELIVER_SM`, `ENQUIRE_LINK`, `BIND_TRANSCEIVER_RESP`, etc.) qui sont renvoyées par la méthode `int getCommandId()`.

Squelette d'un service asynchrone

Le squelette d'un service fonctionnant en mode asynchrone est présenté ci-dessous.

```
01:package org.buguigny.asynchronel;
02:// les imports nécessaires: import ie.omk.smpp. ...
03:
04:public class SMSService extends Thread implements ConnectionObserver {
05:    // Membres privés: Connection, host, port, login, mdp, etc.
```

```
06:    // private static SMSService instance = null; // pour la gestion du singleton
07:
08:    // Un constructeur qui initialise les membres.
09:    // Privé pour un fonctionnement en singleton
10:    private SMSService(String ht, int pt, String un, String pd) {
11:     // ...
12:    }
13:
14:    // Création d'une instance unique de ce service via le constructeur privé
15:    public static SMSService getInstance(String ht, int pt, String un, String pd) {
16:     if(instance == null)
17:         instance = new SMSService(...);
18:     return instance;
19:    }
20:
21:    // Démarrage du thread du service. Cette méthode initialisera la
22:    // connexion au SMS-C, endormira le thread du service jusqu'à la fin du programme.
23:    // Les méthodes du service seront exécutées par un thread de l'API
24:    public void run() {
25:     // cnx = new Connection(..., true);
26:     // cnx.addObserver(this)
27:     // cnx.bind(...);
28:     // wait();
29:    }
30:
31:    // traitement des évènements reçus de l'api
32:    public void update(Connection cnx, SMPPEvent event) {
33:     // switch(event.getType()) {
34:     // case SMPPEvent.RECEIVER_EXIT: ...; break
35:     // case SMPPEvent.RECEIVER_EXCEPTION : ...; break
36:     // case SMPPEvent.RECEIVER_START : ...; break
37:     // }
38:    }
39:
40:    // traitement des paquets SMPP reçus depuis le SMS-C
41:    public void packetReceived(Connection conn, SMPPPacket pdu) {
42:        // switch(pdu.getCommandId()) {
43:     // case SMPPPacket.DELIVER_SM: ...; break;
44:     // case SMPPPacket.BIND_TRANSCEIVER_RESP: ...; break;
45:     // case ... : ...; break;
46:     // }
47:    }
48:}
```

On voit ligne 4 que `SMSService` sera un thread. Ceci permet d'isoler le service du thread principal de l'application afin d'en faciliter la gestion, dans le cas où il y aurait plusieurs services dans une même application ou pour la gestion de la perte de connexion avec le SMS-C.

Le deuxième point à noter est que notre service est un singleton : le constructeur est privé (ligne 10) et la création d'une instance est réalisée par la méthode `SMSService getInstance(...)` (lignes 15 à 19). La raison en est la suivante. Si on pouvait créer plusieurs instances de `SMSService`, on établirait plusieurs connexions avec les mêmes accréditations au SMS-C. Il n'est pas clairement spécifié dans la norme comment le SMS-C doit gérer plusieurs connexions du même service, mais lors des différentes expériences effectuées, on s'aperçoit que lors de l'envoi d'un SMS par un client, les services sont invoqués à tour de rôle (*round robin*), ce qui peut compliquer la gestion des données si celles-ci doivent être partagées entre plusieurs requêtes (par exemple pour un suivi de session). Il est par ailleurs possible qu'un SMS-C donné interdise plusieurs connexions ou, au contraire, qu'il envoie les requêtes simultanément sur l'ensemble des connexions.

La méthode `run()` implémente la réalisation de la connexion au SMS-C. La ligne 26 enregistre notre service auprès de la connexion afin que celle-ci puisse invoquer les méthodes `update()` et `packetReceived` de l'*observer*. Une bonne manière de procéder serait de « sortir » le code correspondant à la gestion de la connexion dans une méthode annexe afin de gérer les ruptures de connexion avec le SMS-C (voir dans l'exemple complet `asynchrone1`). `run()` endort finalement le thread du service : les méthodes de l'observer seront invoquées par un thread de l'API.

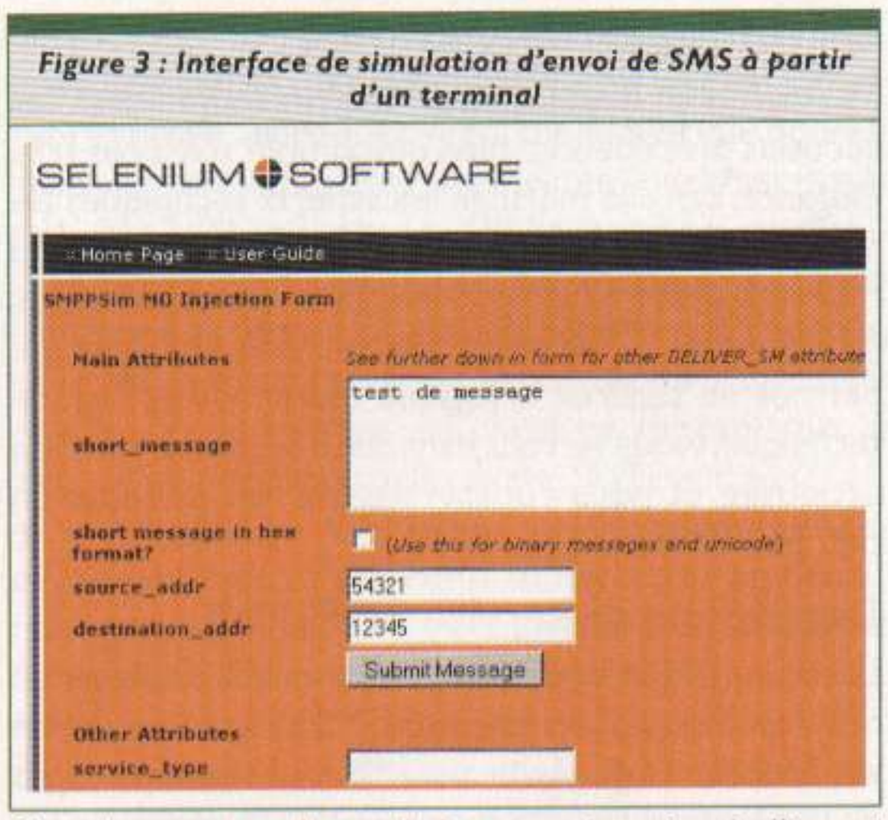
Les évènements relatifs à l'état de la connexion sont communiqués à l'application par la méthode `update()` (lignes 32 à 37). C'est notamment sur réception d'un `RECEIVER_EXIT` qu'on pourra débloquer le thread de service (en appelant la méthode `notify()`) pour sortir de l'application ou pour tenter une reconnexion.

Finalement, toute la logique de service est implémentée dans la méthode `packetReceived()`. A chaque fois qu'un PDU est reçu par la connexion, un `SMPPPacket` est généré et communiqué au service qui doit agir en conséquence. Il ne faut pas oublier de renvoyer des PDU de réponse à chaque requête envoyée par le SMS-C. Dans le cas contraire, on risque soit de bloquer le SMS-C qui serait en attente de trop d'acquittements, soit de provoquer l'envoi d'acquittements négatifs par le SMS-C. Un PDU dont nous n'avons pas encore parlé est `ENQUIRE_LINK` qui peut être envoyé après une certaine période d'inactivité, le plus souvent par le SMS-C, pour vérifier que l'entité à l'autre bout de la connexion est toujours en vie. L'expéditeur s'attend alors à recevoir un `ENQUIRE_LINK_RESP` qui, s'il n'est pas reçu, peut le conduire à fermer la connexion. L'API permet d'automatiser l'envoi des réponses à `ENQUIRE_LINK` si on invoque `autoAckLink(true);` sur l'objet connexion. C'est d'ailleurs le paramétrage par défaut. Il est également possible de demander à l'API d'acquitter automatiquement l'ensemble des requêtes (appel de `autoAckMessages(true);`), mais ce n'est pas conseillé. Nous le ferons cependant pour plus de simplicité.

Traces de fonctionnement d'un service asynchrone

Le répertoire `dev/asynchrone1` contient un exemple d'implémentation de ce squelette. Avant d'analyser son fonctionnement, un petit mot sur le simulateur que nous utilisons. Pour simuler l'envoi d'un SMS par un client, SMPPSim fournit une interface web accessible par l'URL **http://localhost:88/inject_mo.htm**. Un *screenshot* est présenté, sur la figure 3. L'interface offre la possibilité de saisir le texte d'un SMS, les numéros de l'expéditeur et du destinataire. D'autres champs peuvent également être saisis pour être ajoutés au PDU envoyé, mais ne sont pas nécessaires pour notre exemple. Un click sur Submit Message provoque l'envoi du SMS vers l'application.

Figure 3 : Interface de simulation d'envoi de SMS à partir d'un terminal



Si on lance `asynchrone1.jar` et qu'on simule l'envoi d'un SMS, on obtient les traces suivantes :

```
01: DEBUG [main][org.buguigny.asynchrone1.SMSService] getInstance()
02: DEBUG [main][org.buguigny.asynchrone1.SMSService] SMSService()
03: DEBUG [Thread-0][org.buguigny.asynchrone1.SMSService] run()
04: DEBUG [Thread-0][org.buguigny.asynchrone1.SMSService] connect()
05: INFO [Thread-0][org.buguigny.asynchrone1.SMSService] Connection
    au host localhost:2775
06: INFO [Thread-0][ie.omk.smpp.Connection] Creating receiver thread
07: INFO [Thread-0][org.buguigny.asynchrone1.SMSService] Connection
    au host localhost:2775 OK
08: INFO [Thread-0][org.buguigny.asynchrone1.SMSService] Binding au
    SMS-C (localhost:2775). User: smppclient
09: INFO [Thread-0][ie.omk.smpp.Connection] Binding to the SMSC as type 3
10: INFO [Thread-0][ie.omk.smpp.Connection] Opening network link.
11: INFO [Thread-0][ie.omk.smpp.net.TcpLink] Opening TCP socket to
    localhost/127.0.0.1:2775
12: INFO [Thread-0][ie.omk.smpp.Connection] Setting state 1
13: INFO [Thread-0][org.buguigny.asynchrone1.SMSService] Bind au
    SMS-C (localhost:2775). User: smppclient OK
14: INFO [ReceiverDaemon][ie.omk.smpp.Connection] Receiver thread started
15: INFO [ReceiverDaemon][org.buguigny.asynchrone1.SMSService] Reçu
    evenement: RECEIVER_START: ie.omk.smpp.event.ReceiverStartEvent@a3736
16: INFO [ReceiverDaemon][ie.omk.smpp.Connection] Setting state 2
17: WARN [ReceiverDaemon][ie.omk.smpp.Connection] Disabling optional
    parameter support as no sc_interface_version parameter was received
18: INFO [ReceiverDaemon][ie.omk.smpp.Connection] Notifying observers
    of packet received
19: INFO [ReceiverDaemon][org.buguigny.asynchrone1.SMSService] Reçu
    Message d'acquittement au Bind
20: INFO [ReceiverDaemon][ie.omk.smpp.Connection] deliver_sm_resp sent.
21: INFO [ReceiverDaemon][ie.omk.smpp.Connection] Notifying observers
    of packet received
22: INFO [ReceiverDaemon][org.buguigny.asynchrone1.SMSService] Reçu
    Message SMS: 'test de message'
```

Les traces sont obtenues par Log4J. Nous verrons plus tard où et comment les paramètres de trace sont positionnés. Les informations tracées sont la criticité, le nom du thread, la classe et un message. L'initialisation de l'application est faite entre les lignes 1 à 19 et l'envoi du SMS entre les lignes 20 et 22. La création du service est faite dans les lignes 1 et 2 par le thread `main` qui donne ensuite la main au thread `Thread-0` qui correspond à notre service. Dans les lignes 3 à 13, on peut suivre la création de la connexion que ce soit au niveau de l'API (traces des classes `ie.omk.smpp.*`) qu'au niveau de l'application (classe `org.buguigny.asynchrone1.SMSService`). Le thread `Thread-0` est finalement endormi dans la méthode `run()` et un thread de l'API, `ReceiverDaemon`, prend ensuite la main pour gérer la connexion et notifier au service des différents évènements. Ainsi, lorsqu'elle reçoit un message, la connexion (et donc `ReceiverDaemon`) informe l'ensemble des observers qui se sont enregistrés (nous n'en avons qu'un seul, `SMSService`) en appelant `packetReceived()` (lignes 18, 19 et 21, 22). Notons que nous avons positionné l'acquittement à toutes les requêtes à vrai. Ligne 20 correspond à l'acquittement automatique du PDU `DELIVER_SM`.

Faisons une expérience. Dans un mode de fonctionnement normal, le SMS-C tourne en permanence, l'application s'y connecte et les échanges de messages peuvent commencer. Lançons le simulateur dans une fenêtre et l'application dans une autre. Tuons ensuite le simulateur (Control-C), ce qui dans un environnement de production correspondrait au plantage du SMS-C. On verrait sur les traces que le thread `Thread-0` est réveillé et que la phase de connexion recommence. On observerait un comportement semblable si on tentait de lancer l'application alors que le simulateur n'était pas encore lancé. Ce comportement n'est pas pris en charge par l'API, mais par l'application elle-même.

Faisons une autre expérience. Pour simuler un certain temps de traitement lors de la réception d'un SMS (par exemple un accès à une base de données), la gestion du PDU `DELIVER_SM` dans le service met en place une boucle d'attente de 0 à 10000 (voir fichier `SMSService.java`, méthode `packetReceived`). Pour simuler une forte charge sur le SMS-C, lancez le simulateur, envoyez quelques SMS (une dizaine) par l'interface web du simulateur, puis, seulement après, lancez l'application. Ci-dessous sont données les traces avec l'information sur l'heure d'exécution.

```
08:05:25,944 INFO [ReceiverDaemon][org.buguigny.
      asynchrone1.SMSService] Reçu Message ...
08:05:26,945 INFO [ReceiverDaemon][org.buguigny.
      asynchrone1.SMSService] Reçu Message ...
08:05:27,947 INFO [ReceiverDaemon][org.buguigny.
      asynchrone1.SMSService] Reçu Message ...
08:05:28,948 INFO [ReceiverDaemon][org.buguigny.
      asynchrone1.SMSService] Reçu Message ...
08:05:29,970 INFO [ReceiverDaemon][org.buguigny.
      asynchrone1.SMSService] Reçu Message ...
```

On observe que c'est toujours le même thread, `ReceiverDaemon`, qui exécute la méthode de notification au service. L'API fonctionne dans un mode asynchrone, mais ne fait tourner qu'un seul thread ! Le ralentissement de traitement est du à la boucle de simulation de charge dans le service. Il est évident que dans un environnement de production où on peut être amené à gérer 10, 50, voire plus de 1000 SMS par seconde, ce type de fonctionnement n'est pas acceptable.

L'API nous offre une solution à ce problème. Dans son architecture interne, elle définit deux

classes pour la distribution des évènements : `SimpleEventDispatcher` et `ThreadedEventDispatcher`. `SimpleEventDispatcher` est *monothreadé*. Il est utilisé par défaut. `ThreadedEventDispatcher` est quant à lui *multithreadé*, mais, attention, la documentation nous informe qu'il est hautement expérimental. Lors de son initialisation, `ThreadedEventDispatcher` crée un pool de threads et pour chaque évènement à notifier, un thread différent invoque `packetReceived` ou `update()`. La mise en place de ce dispatcher ainsi que son paramétrage sont réalisés dans le fichier de configuration `smppapi.properties` que nous verrons plus bas. Pour terminer, voici les traces obtenues avec le `ThreadedEventDispatcher` où on voit que l'ensemble des requêtes est traité dans la même fraction de seconde.

```
08:36:31,897 INFO [EventDispatch2][org.buguigny.
        asynchrone1.SMSService] Reçu Message ...
08:36:31,897 INFO [EventDispatch3][org.buguigny.
        asynchrone1.SMSService] Reçu Message ...
08:36:31,897 INFO [EventDispatch4][org.buguigny.
        asynchrone1.SMSService] Reçu Message ...
08:36:31,897 INFO [EventDispatch5][org.buguigny.
        asynchrone1.SMSService] Reçu Message ...
08:36:31,907 INFO [EventDispatch6][org.buguigny.
        asynchrone1.SMSService] Reçu Message ...
```

Fichiers de configuration

Pour fonctionner, `smppapi` a besoin de deux fichiers de configuration. Le premier, `log4j.properties`, définit les paramètres des traces de l'API. On peut également y ajouter les paramètres de trace du service (voir fichier `dev/asynchrone1/src/etc/log4j.properties`). Le second fichier, `smppapi.properties`, définit les paramètres de fonctionnement de l'API elle-même. Ces paramètres sont documentés dans le javadoc de la classe `APIConfig`. Les principaux sont (voir le fichier `dev/asynchrone1/src/etc/amppapi.properties` pour un exemple complet) :

- `smppapi.net.buffersize_in` et `smppapi.net.buffersize_out` spécifient les tailles des *buffers* d'entrées/sorties. La valeur peut être spécifiée en octets (i. e. :4096 pour 4Ko) ou en kilo-octets (i. e. :4k).
- `smppapi.net.autoflush` spécifie si un *flush* doit être envoyé à chaque fois qu'un paquet est envoyé. La valeur est `true` ou `false`.
- `smppapi.connection.rcv_daemon.ioex_count` spécifie le nombre d'exceptions d'entrées/sorties après lequel la connexion enverra un évènement `RECEIVER_EXIT`.
- `smppapi.event.dispatcher` positionne le type d'*event dispatcher* à employer. La valeur peut être `SimpleEventDispatcher` ou `ThreadedEventDispatcher` (ou un dispatcher de votre cru...). Dans le cas du `ThreadedEventDispatcher`, deux paramètres additionnels doivent être positionnés :
  - `smppapi.event.threaded_dispatcher.pool_size` définit la taille du pool de threads.
  - `smppapi.event.threaded_dispatcher.queue_size` définit la taille de la file FILO du dispatcher.

Développement avancé

La gestion en mode asynchrone présentée dans les sections précédentes, bien que simple, n'est pas très élégante, car elle mélange les aspects techniques (le `switch` avec les différents évènements internes ou réseau, la gestion des reconnexions) avec des aspects fonctionnels. L'API nous fournit une solution qui permet de séparer la logique métier de la logique technique. Nous verrons dans cette section la solution proposée et nous l'utiliserons sur un exemple un peu plus complexe.

Description du service SMS2Mail

Le service SMS2Mail permet à un abonné **A** d'envoyer un mail à un abonné **B** via le service SMS. Le fonctionnement est le suivant : **A** envoie un SMS à un numéro court de l'opérateur. Le SMS devra avoir le format suivant : <no de tel de **B**><espace><message>. Si **A** a souscrit au service (ce qui suppose que l'opérateur connaît son adresse mail), si le numéro de **B** est correct (le numéro est syntaxiquement correct et l'abonné existe) et si **B** a souscrit également au service (on connaît également son adresse), SMS2Mail enverra le mail contenant <message> à **B** et enverra un SMS de confirmation à **A**. Dans le cas contraire, le service doit envoyer un SMS d'erreur à **A**. Les sources complets de ce service se trouvent dans le répertoire `sms2mail`.

D'un point de vue technique, le service doit être capable de recevoir et d'envoyer des SMS. La connexion devra donc fonctionner en mode transceiver. Par ailleurs, l'analyse de la structure du SMS, mais surtout l'envoi de mail ne sont pas immédiats : pour traiter plusieurs SMS en parallèle, il est grandement préférable d'utiliser le dispatcher multithreadé. Finalement, la logique métier décrite ci-dessus devra être implémentée dans la partie gérant l'évènement `DELIVER_SM`, puisque le service est déclenché sur réception de SMS. Si on applique le *template* de squelette présenté auparavant, la classe `SMSService` contiendrait la gestion des connexions et reconnexions, le `switch` de gestion des évènements et, en plein milieu du `switch`, la logique métier, ce qui n'est optimal ni pour la mise au point du service ni pour sa maintenance future. La solution consisterait à définir une classe `ServiceLogic` dont des méthodes spécifiques seraient appelées en fonction des évènements reçus. Ainsi, notre logique métier pourrait être isolée dans une méthode qui pourrait s'appeler `traiteReceptionDELIVER_SM`. C'est à peu de chose près la solution proposée par l'API.

La classe SMPPEventAdapter

La classe `SMPPEventAdapter` permet une gestion plus simple des évènements produits par l'API, qu'il s'agisse de d'évènements internes, traités dans l'observer dans `update()`, ou d'évènements réseau, traités par `packetReceived()`. Elle définit une méthode par type d'évènement reçu. Ces méthodes ont une signature générique, `void nomMethode(Connection cnx, <ObjetEvenement> evt)` et sont vides par défaut. Ainsi, la méthode appelée lors de la réception d'un paquet

DELIVER_SM est public void deliverSM(Connection cnx DeliverSM evt), où DeliverSM est un objet représentant le paquet reçu sur lequel on pourra appliquer les méthodes evt.getSource().getAddress(), evt.getDestination().getAddress() et evt.getMessageText() pour récupérer respectivement l'adresse (i. e. : le numéro de téléphone) de l'expéditeur, celle du destinataire et le texte du message SMS envoyé. Pour mettre en place ce mécanisme, il faut définir une sous-classe de SMPPEventAdapter et surcharger les méthodes correspondantes aux évènements qu'on souhaite traiter. Pour terminer, notons que SMPPEventAdapter implémente l'interface ConnectionObserver, mais les méthodes update() et packetReceived() sont non surchargeables, car déclarées en final. Elles sont chargées d'invoquer les méthodes de gestion des évènements.

Structure du service SMS2Mail

Avec tous ces éléments en main, notre service sera implémenté de la manière suivante. Nous allons implémenter dans la classe SMS2MailService les aspects purement techniques : création de la connexion et gestion des pertes du lien. La classe SMS2MailServiceLogic, sous-classe de SMPPEventAdapter, se situe entre le technique et le fonctionnel purs et est responsable de la réception et de l'envoi de SMS. Finalement, MailSender est en charge de la logique métier à proprement parler.

SMS2MailService est très proche de la classe SMSService que nous avons définie dans le squelette asynchrone1. La différence principale est que ce n'est plus SMSService qui est l'observer, mais SMS2MailServiceLogic. Comme précédemment, SMS2MailService est un thread qui reste bloqué dans la méthode run() en attente d'être réveillée en cas de perte de connexion. La perte de connexion sera détectée par l'API et le service en sera informé par l'invocation des méthodes receiverExit() et receiverExitException(). Afin qu'elles puissent à leur tour invoquer notify() sur le bon objet, une référence sur SMS2MailService doit être définie dans SMS2MailServiceLogic.

La réception de SMS et l'envoi de SMS de notification au client **A** sont implémentées dans la méthode deliverSM() de SMS2MailServiceLogic. Les contraintes purement métier (vérification de la souscription au service, de l'existence du client **B**, envoi effectif du mail, etc.) sont définies dans la méthode sendMail() de MailSender. Lors de la réception d'un SMS, deliverSM() extrait le numéro de téléphone de l'expéditeur ainsi que son message, crée un objet MailSender et appelle sendMail(). En fonction du résultat, elle envoie par SMS la notification correspondante à **A**.

MailSender simule les accès en base de données client en définissant une table statique de clients, baseClient. On supposera qu'un client présent dans baseClient et pour lequel l'adresse mail est différente de null est un client qui a souscrit au service. On suppose également que le plan de numérotation de notre opérateur fictif définit les numéros de téléphone au format Ø1xxxx. La méthode sendMail() réalise les opérations suivantes et renvoie « 0 » en cas de succès ou un code d'erreur :

- analyse du format du message qui doit être de la forme <no de tel de **B**><espace><message> ;
- vérification que **A** a souscrit au service ;
- vérification que **B** est un client qui existe dans la base et qu'il a souscrit au service ;
- simulation de l'envoi du mail. Le succès ou l'échec sont simulés en tirant à pile ou face.

Vous pouvez maintenant essayer le service avec différents types de clients, ceux définis dans la base ayant ou non souscrit au service et des clients dont le numéro n'existe pas. Suivez les traces de l'application et du simulateur dans leurs fenêtres respectives.

Conclusion

Nous avons présenté dans cet article les bases nécessaires au développement d'applications SMS en Java. Bien entendu, tous les paramétrages et toutes les possibilités n'ont pu être abordés ici, mais il ne faut jamais oublier que pour bien prendre en main une technologie, il faut jouer avec, essayer les différentes options, analyser ses erreurs. Amusez-vous bien, car les possibilités sont grandes.

Maurice Szmurlo,

maurice.szmurlo@buguigny.org

RÉFÉRENCES

- **[1]** Signaling System 7 (SS7) : **http://www.iec.org/online/tutorials/ss7/**
- **[2]** SS7 Tutorial : **http://www.pt.com/tutorials/ss7/index.html**
- **[3]** LAGRANGE (Xavier), GODLEWSKI (Philippe), TABBANE (Sami), *Réseaux GSM*, Hermes Science Publications, 5ème édition, 2000.
- **[4]** *Short Message Peer to Peer protocol specification* : **http://www.smsforum.net/**
- **[5]** Kannel : *Open Source WAP and SMS Gateway* (**www.kannel.org**) : **http://www.kannel.org/**
- **[6]** Logica : SMPP Software : **http://opensmpp.logica.com/**
- **[7]** Selenium Software : simulateur de SMS-C : **http://www.seleniumsoftware.com/**
- **[8]** **http://smppapi.sourceforge.net/**
- **[9]** International Telecommunication Union, Q.931 (05/98) Spécification de la couche 3 de l'interface utilisateur-réseau RNIS pour la commande de l'appel de base : **http://www.itu.int/rec/T-REC-Q.931/fr/**

Yann Langlais

Jouer avec la « libdl »

En 1992, alors que je découvrais les joies de l'édition de texte avec Emacs, une question m'est venue à l'esprit. Pourquoi diable les extensions d'Emacs ont-elles été écrites en Elisp (Emacs-lisp) et non en C ? La réponse était simple. Il n'existait pas alors de possibilité standard en C de charger et décharger des bibliothèques en cours d'exécution.
Quelques années plus tard, et bien qu'Emacs ait été remplacé par Vi dans mes habitudes dactylographiques, je découvris avec plaisir un outil qui rendait caduque mon explication. Je venais de tomber par hasard sur dlfcn.h.

Ce fichier d'en-tête définit les prototypes de points d'entrée forts appréciables et permettant de charger et décharger des bibliothèques et de récupérer des points d'entrée ou des symboles (variables) pour les utiliser à volonté.

Aujourd'hui, et bien que ces fonctions soient à la base de la notion des « plug-in » en C et C++, leur utilisation me semble encore trop restreinte. Voyons à travers quelques exemples ce que nous pouvons en faire.

Les exemples fournis n'ont pas pour vocation d'être toujours justes ni d'effectuer tous les tests appropriés. Ils n'ont valeur d'exemple que pour le domaine très restreint de l'illustration du propos. En temps normal, il est en particulier nécessaire de tester toutes les valeurs retour de `dlopen` et `dlsym` avant de les utiliser.

L'article est basé sur Solaris et Linux.

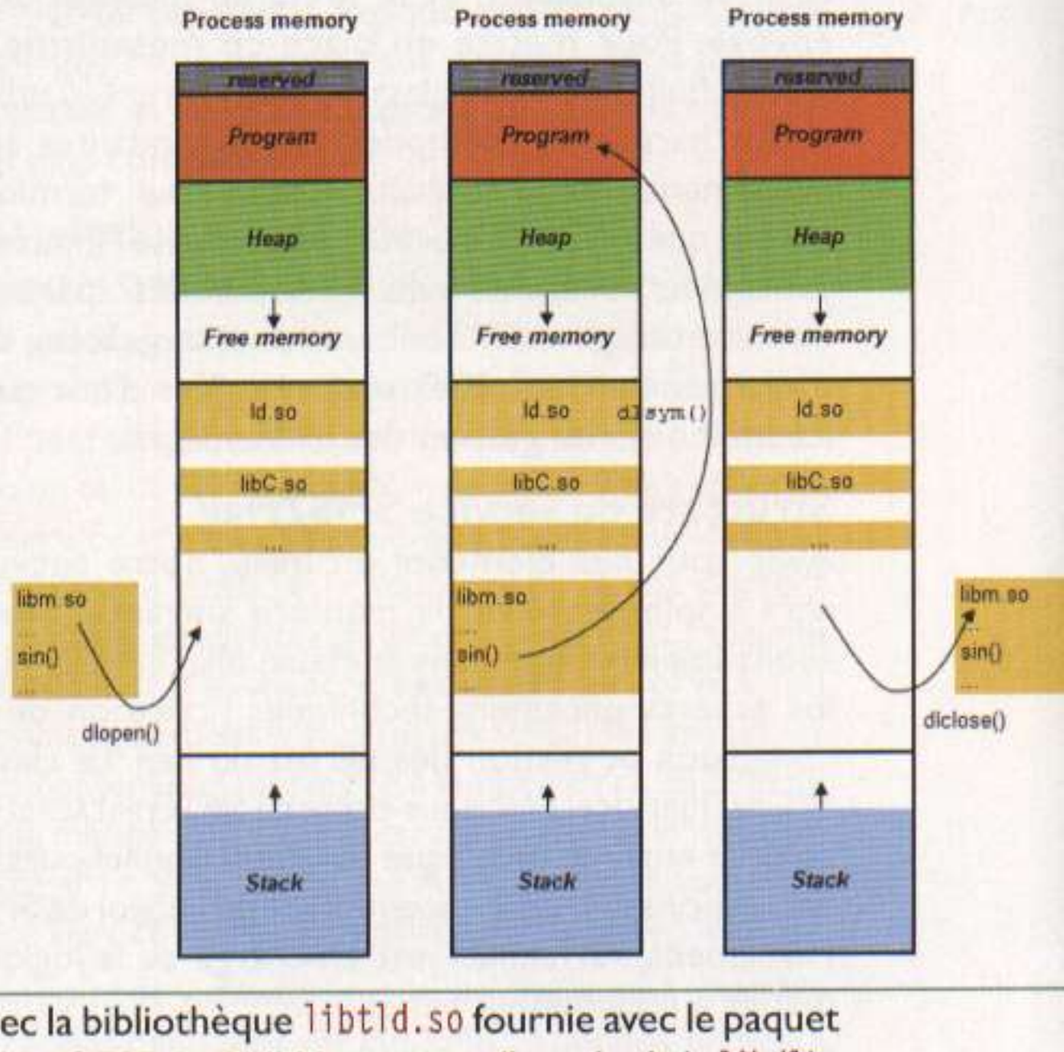
1. Présentation de la libdl.so

1.1 Présentation générale

La `libdl.so` permet de charger (et de décharger) des bibliothèques partagées pendant l'exécution d'un programme et de mettre à disposition du programme tout ou partie des symboles (noms de fonctions ou de variables globales) chargés. C'est un éditeur de liens en cours d'exécution. Il effectue les mêmes tâches que `ld` lors de l'étape de l'édition de liens (figure 1).

Cette bibliothèque est intimement liée au format de fichiers binaires ELF **[1]** (*Extecutable and Linkable Format*) qui a permis d'obtenir du code librement déplaçable dans l'espace mémoire des processus. Il existe des plateformes qui ne supportent pas le format ELF. Il existe aussi des plateformes qui, bien que supportant les bibliothèques partagées, n'implémentent pas la `libdl`. Pour une partie de ces plateformes, il existe cependant une alternative avec la bibliothèque `libtld.so` fournie avec le paquet `libtool` **[2]** de GNU. Les points d'entrée de la `libdlt.so` sont similaires à ceux de la `libdl.so`, mais sont préfixés par « `lt_` ».

Figure 1 : Synoptique du fonctionnement de dlopen, dlsym et dlclose



Les variations entre les différentes plateformes viennent du relatif jeune âge de cette bibliothèque et donc de sa standardisation récente et imparfaite.

Initialement créée par SUN, la norme POSIX 1003.1 **[3]** a standardisé les fonctions `dlclose`, `dlerror`, `dlopen` et `dlsym`. Ces fonctions sont aussi présentes dans la Linux Standard Base (LSB) version 1.3 **[4]**. Sun a, par la suite, ajouté la fonction `dladdr`. GNU l'a reprise et a ajouté `dlvsym`.

Les différents drapeaux `RTLD_*` ne sont pas non plus tous standardisés, mais, au moins sur Solaris et Linux, les valeurs engendrent des comportements assez proches les uns des autres (à une exception importante près que nous verrons au paragraphe « Chaînage de fonctions »).

1.2 Description des points d'entrée

La bibliothèque contient 5 points d'entrée principaux : `dlopen`, `dlsym`, `dlclose`, `dlerror` et `dladdr`.

1.2.1 dlopen

```
extern void *   dlopen(const char *, int);
```

Cette fonction charge en mémoire la bibliothèque désignée par la chaîne de caractère passée en premier paramètre. Le second paramètre est un drapeau qui fixe le mode de fonctionnement de l'éditeur de liens à la volée. Ce second peut prendre les valeurs (documentées) suivantes :

- `RTLD_LAZY`, pour ne résoudre les symboles de la bibliothèque à ouvrir que lorsque ceux-ci sont explicitement demandés ;
- `RTLD_NOW`, pour résoudre tous les symboles lors du chargement de la bibliothèque.

En complément de ces valeurs, il existe une co-valeur `RTLD_GLOBAL` augmentant la portée des symboles en les rendant disponibles externes (i. e. non préfixés en C par `static`) pour les bibliothèques chargées par la suite. Elle s'emploie en l'accolant par un `ou` binaire « `|` » à l'une des valeurs possibles du drapeau : `RTLD_LAZY | RTLD_GLOBAL` ou `RTLD_NOW | RTLD_GLOBAL`.

`dlopen` retourne un pointeur vers une poignée (*handle*) caractéristique de la bibliothèque sous forme d'un pointeur générique (`void *`). Celui-ci sera nul en cas d'échec.

Notons que si `dlopen` renvoie `NULL`, la variable globale `errno` ne devrait pas avoir été modifiée et que `strerr()` ne pourra donc fournir d'explication correcte. Il faudra pour cela appeler `dlerror()`.

1.2.2 dlsym

```
extern void *   dlsym(void *, const char *);
```

`dlsym` se charge de retrouver (résoudre) l'adresse d'un symbole dont le nom est passé en paramètre.

Le premier paramètre passé est un pointeur vers la « poignée » retournée par `dlopen`.

Il existe en standard deux poignées particulières définies lorsque `_GNU_SOURCE` est définie. Ces poignées sont définies par les symboles `RTLD_DEFAULT` et `RTLD_NEXT`.

`RTLD_DEFAULT` précise à `dlsym` qu'il faut rechercher la première occurrence du symbole dans les bibliothèques déjà chargées en mémoire, dans l'ordre de leur chargement.

`RTLD_NEXT` stipule à `dlsym` qu'il lui faut chercher l'occurrence suivante du symbole dans les bibliothèques suivant la bibliothèque en cours. Ce pseudo-pointeur ne fonctionne que pour les bibliothèques partagées.

Suivant les implémentations, on pourra aussi retrouver le symbole `RTLD_SELF` qui recherchera le symbole dans la bibliothèque en cours.

Le second paramètre est le symbole à rechercher.

`dlsym` retourne le pointeur sur le symbole demandé, ou, si celui-ci n'est pas retrouvé, un pointeur `NULL`.

ATTENTION

`dlsym()` ne permet pas de résoudre que des noms de fonction, mais tous les symboles en général. On peut donc aussi retrouver des variables globales (i. e. variables définies hors fonctions et sans l'attribut `static`).

1.2.3 dlclose

```
extern int      dlclose(void *);
```

La fonction `dlclose()` a pour fonction d'éliminer les liens et de décharger des bibliothèques chargées par `dlopen()`.

Elle prend en argument la poignée fournie par `dlopen()`.

Elle retourne `0` quand tout ce passe bien ou un code d'erreur positif en cas de soucis.

1.2.4 dlerror

```
extern char *   dlerror(void);
```

`dlerror()` retourne une chaîne de caractères contenant le dernier message d'erreur généré par la `libdl` ou `NULL` si aucune erreur n'est survenue.

Après l'appel, `dlerror()` remet les codes d'erreur de la bibliothèque à 0. En conséquence, un second appel à `dlerror` suivant immédiatement le premier renverra `NULL`.

Les chaînes de caractères retournées ne doivent pas être libérées par `free()`.

1.2.5 dladdr

```
extern int      dladdr(void *, Dl_info *);
```

`dladdr()` détermine si une adresse est localisée dans un objet en mémoire. Si tel est le cas, il retourne des informations relatives à cet objet.

`dladdr()` prend en premier paramètre une adresse sous la forme d'un pointeur anonyme (`void *`). C'est cette adresse qu'il essaiera de retrouver parmi les segments de texte et de données présents en mémoire. Le second paramètre est un pointeur vers une structure allouée de type `DL_info` qui sert de réceptacle aux informations de retour. La `DL_info` contient les champs suivants :

```
typedef struct      dl_info {
        char                *dli_fname;
        void                *dli_fbase;
        char                *dli_sname;
        void                *dli_saddr;
} Dl_info;
```

`dli_fname` contient le nom du fichier dans lequel est stocké le pointeur fourni en argument ; `dli_fbase` contient l'adresse de base du fichier en mémoire ; `dli_sname` contient le nom du fichier source du bloc correspondant au pointeur fourni si les informations de *debug* sont présentes. `dli_sbase` contient l'adresse du symbole contenant le pointeur fourni.

`dladdr()` retourne 0 s'il ne trouve pas d'objet contenant le pointeur passé. Toute autre valeur que 0 signifie que des informations ont été trouvées et que la structure `DL_info` a été remplie.

Pour de plus amples informations sur ces points d'entrée, il est nécessaire de s'en remettre aux manuels **[5]**.

D'autres informations concernant la programmation de bibliothèques partagées peuvent être trouvées en référence **[6]**.

Enfin, certaines variables d'environnement influent sur les bibliothèques, ou plutôt sur l'éditeur de liens **[7]**. L'une d'entre elles sera particulièrement utile dans les paragraphes suivants.

1.2.6 Constructeur et destructeur

Il est parfois nécessaire d'accomplir certaines actions lors du chargement ou du déchargement d'une bibliothèque. On peut, par exemple, devoir réserver de la mémoire et initialiser certaines variables lors du chargement, et libérer la mémoire avant le déchargement.

Pour ce faire, il n'y a malheureusement pas de norme, mais deux écoles distinctes. Il y a, tout d'abord, la méthode « historique » qui est de définir deux fonctions `void _init()` et `void _fini()`. Ces 2 points d'entrée sont, en général, créés automatiquement par les différents *linkers*. Le fait de les redéfinir dans nos bibliothèques et de demander au linker d'éviter de les générer permet donc de remplacer les fonctions standards par les nôtres. Seulement, en général, les éditeurs de liens ne génèrent pas des fonctions vides et le code manquant risque d'avoir des effets secondaires indésirables. Du reste, si Linux supporte encore la fonctionnalité en tant qu'« obsolète », Sun l'interdit (il ne permet pas de demander au compilateur de ne pas générer la fonction standard). Il est donc fortement recommandé d'éviter l'utilisation de ces fonctions.

La seconde méthode revient à utiliser des directives de compilation propriétaires. Par exemple, si l'on veut que la fonction `void onload(void)` soit appelée au chargement de la bibliothèque, et `void onunload(void)`, on écrira sous Linux (compilateur gcc) dans le code source :

```
void __attribute__ ((constructor)) onload(void) {
        printf("loading\n");
}

void __attribute__ ((destructor)) onunload(void) {
        printf("unloading\n");
}
```

et sous Sun (compilateur Forte 6) :

```
#pragma init (onload)
void onload(void) {
        printf("loading\n");
}

#pragma fini (onunload)
void onload(void) {
        printf("unloading\n");
}
```

Il est clair que si l'on recherche la portabilité, il est nécessaire de faire en sorte de se passer de ce genre de chose, à moins d'utiliser les `#ifdef`.

2. Un exemple simple

Voyons l'utilisation de la `libdl` à travers un petit exemple simple.

Admettons que nous voulions écrire un programme qui fournit le sinus d'une liste de valeurs passées en paramètre. Admettons de plus que nous n'ajoutions pas la bibliothèque `libm` à la compilation.

Nous allons donc charger notre bibliothèque « manuellement » pendant l'exécution du programme.

```
cat > sin.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>
int main(int n, char *a[]) {
    int i;
    void *lib;
    double (*sin)(double);

    /* load /usr/lib/libm.so into memory: */
    if (!(lib = dlopen("/usr/lib/libm.so", RTLD_
LAZY)))
        return 1;

    /* resolve symbole "sin": */
    if (!(sin = (double (*)(double)) dlsym(lib,
"sin")))
        return 2;

    /* print sine of all arguments: */
    for (i = 1; i < n; i++) printf("sin(%f) = %f\n",
        atof(a[i]), sin(atof(a[i])));

    /* unload /usr/lib/libm.so */
    dlclose(lib);

    return 0;
}
EOF
% gcc sin.c -o sin -ldl
```

Le lancement du programme `sin` donne :

```
% sin 0 0.22 .5 1 3.14159
sin(0.000000) = 0.000000
sin(0.220000) = 0.218230
sin(0.500000) = 0.479426
sin(1.000000) = 0.841471
sin(3.141590) = 0.000003
```

Nous pouvons aussi rendre plus « générique » notre programme en lui faisant calculer autre chose que des sinus. Nous pouvons, par exemple, avec un unique exécutable, faire calculer de nombreuses fonctions mathématiques.

Pour ce faire, nous pouvons, par exemple, tenter de charger la fonction qui porte le même nom que le programme (argument 0). Ainsi, en renommant le programme ou en faisant des liens symboliques vers d'autres symboles, notre programme aura des comportements différents :

```
% cat > math.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>
int main(int n, char *a[]) {
    int i;
    void *lib;
    double (*math)(double);

    /* load /usr/lib/libm.so into memory: */
    if (!(lib  = dlopen("/usr/lib/libm.so", RTLD_LAZY)))
        return 1;
    /* resolve symbole a[0]: */
    if (!(math = (double (*)(double)) dlsym(lib, a[0])))
        return 2;
    /* call a[0] for all command line agruments: */
    for (i = 1; i < n; i++) printf("%s(%f) = %f\n",
        a[0], atof(a[i]), math(atof(a[i])));
    /* unload libm: */
    dlclose(lib);

    return 0;
}
EOF
% gcc math.c -o sin -ldl
```

et voyons le résultat :

```
% sin 0 0.22 .5 1 3.14159
sin(0.000000) = 0.000000
sin(0.220000) = 0.218230
sin(0.500000) = 0.479426
sin(1.000000) = 0.841471
sin(3.141590) = 0.000003
% mv sin cos
% cos 0 0.22 .5 1 3.14159
cos(0.000000) = 1.000000
cos(0.220000) = 0.975897
cos(0.500000) = 0.877583
cos(1.000000) = 0.540302
cos(3.141590) = -1.000000
% ln -s cos tan
tan 0.22 .5 1 3.14159
tan(0.220000) = 0.223619
tan(0.500000) = 0.546302
tan(1.000000) = 1.557408
tan(3.141590) = -0.000003
% mv tan log
log 1 2.718282 10
log(1.000000) = 0.000000
log(2.718282) = 1.000000
log(10.000000) = 2.302585
% cp cos sqrt
sqrt 2 4 9
sqrt(2.000000) = 1.414214
sqrt(4.000000) = 2.000000
sqrt(9.000000) = 3.000000
```

Bien sûr, cet exemple ne fonctionne que pour les fonctions de la bibliothèque mathématique `libm` qui ont pour prototype `double (*)(double)`. Si l'on renommait notre programme original en `pow`, le résultat serait tout à fait aléatoire : la fonction `pow` de la `libm` ira chercher un second argument sur la pile et prendra ce qu'il trouvera.

3. Portée des symboles : le privé et le public

La portée des symboles d'une bibliothèque varie en fonction de la manière dont sont définis les symboles. Seuls les symboles définis en dehors d'une fonction sont accessibles de l'extérieur. La notion de « portée » est déterminée par les mots `static` et `extern` (ou rien). Voyons cela à travers un exemple :

```
% cat > libtest.c <<EOF
static int static_int = 55;
       int extern_int = 66;
static void static_function() {
    extern_int--;
}
void extern_function() {
    extern_int++;
}
EOF
% gcc -G -shared -fPIC libtest.c -o libtest.so
% cat > portee.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
int main(void) {

    void *h;
    void *p;
    if (!(h = dlopen("libtest.so", RTLD_LAZY))) {
        fprintf(stderr, "error loading libtest.so: %s\n",
                dlerror());
        return 1;
    }
    if (!(p = dlsym(h, "static_int")))
        fprintf(stderr, "error resolving static_int: %s\n",
                dlerror());
    else
        printf("libtest.static_int is at %x "
                "and its value is %d\n", p, *(int *) p);

    if (!(p = dlsym(h, "extern_int")))
        fprintf(stderr, "error resolving extern_int: %s\n",
                dlerror());
    else
        printf("libtest.extern_int is at %x "
                "and its value is %d\n", p, *(int *) p);

    if (!(p = dlsym(h, "static_function")))
        fprintf(stderr,
                "error resolving static_function: %s\n",
                dlerror());
    else
        printf("libtest.static_function is at %x\n", p);

    if (!(p = dlsym(h, "extern_function")))
        fprintf(stderr,
                "error resolving extern_function: %s\n",
                dlerror());
    else
        printf("libtest.error_function is at %x\n", p);

    dlclose(h);
    return 0;
}
EOF
% cc -o portee portee.c -ldl
```

Vérifions tout d'abord la portée (`Bind`) des symboles avec `nm` (sous Solaris, le résultat de `nm` sous Linux est un peu moins parlant) :

```
% nm libtest.so | egrep \
"Index|static_int|extern_int|static_function|extern_function"
[Index]  Value Size Type  Bind  Other Shndx  Name
[41]     |  672|  36|FUNC |GLOB |0    |5     |extern_function
[42]     |66404|   4|OBJT |GLOB |0    |10    |extern_int
[31]     |  616|  36|FUNC |LOCL |0    |5     |static_function
[30]     |66408|   4|OBJT |LOCL |0    |10    |static_int
```

On voit clairement que les symboles préfixés par `static_` et définis localement n'ont qu'une portée locale (`Bind` vaut `LOCL` par opposition à `GLOB` pour « globale »). Voilà ce que l'on obtient en lançant le programme de test :

```
% portee
error resolving static_int: ld.so.1:
 portee: fatal: static_int: can't find symbol
libtest.extern_int is at ff270364 and its value is 66
error resolving static_function: ld.so.1:
 portee: fatal: static_function: can't find symbol
libtest.error_function is at ff2602a0
```

Dans cet exemple, la seule manière d'accéder à `static_int` depuis notre programme de test serait d'ajouter des fonctions spécifiques :

```
% cat > libtest2.c <<EOF
static int static_int = 55;
       int extern_int = 66;
```

```
static void static_function() {
    extern_int--;
}

void extern_function() {
    extern_int++;
}
int  static_int_get() {
    return static_int;
}
int  static_int_set(int i) {
    return static_int = i;
}
void static_int_print() {
    printf("static_int = %d\n", static_int);
}
EOF
% gcc -G -shared -fPIC libtest2.c -o libtest2.so
% cat > portee2.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
int main(void) {
    void *h;
    void *p;
    int  (*get)();
    int  (*set)();
    void  (*print)();

        if (!(h = dlopen("libtest2.so", RTLD_LAZY))) {
        fprintf(stderr, "error loading libtest2.so: %s\n",
                dlerror());
        return 1;
    }
}
if (!(get = (int (*)()) dlsym(h, "static_int_get")))
    fprintf(stderr, "error resolving static_int: %s\n",
            dlerror());
else
    printf("static_int_get() returns %d\n", get());

if (!(set = (int (*)(int)) dlsym(h, "static_int_set")))
    fprintf(stderr,
            "error resolving static_int_set: %s\n",
            dlerror());
else
    printf("setting static_int with "
            "static_int_get() to 33\n", set(33));

if (!(print = (void (*)()) dlsym(h, "static_int_print")))
    fprintf(stderr,
            "error resolving static_int_print: %s\n",
            dlerror());
else {
    printf("call static_int_print() : ");
    print();
  }
  dlclose(h);
  return 0;
}
EOF
% gcc portee2.c -o portee2 -ldl
```

Le nouveau programme de test donne :

```
% portee2
static_int_get() returns 55
setting static_int with static_int_get() to 33
call static_int_print() : static_int = 33
```

Ce genre de restrictions et ces formes d'appels ressemblent bien à celle que l'on peut avoir avec les attributs `private` ou `public` du C++.

4. Vers l'auto-programmation

Le mythe de l'auto-programmation n'est pas récent. Dans la philosophie du `lisp`, par exemple, le programme est une liste et la liste est [potentiellement] un programme. Tout programme est donc capable de se créer des extensions. En règle générale, les langages de script sont des instruments de choix pour l'auto-programmation ou la génération de code spécifique conditionnel et son évaluation à la volée :

```
#!/bin/zsh
cmd="ls -alrt $1"
eval $cmd
```

Le BASIC des années 80 avait sa commande `merge` qui permettait de créer du code dans un fichier et de le réinjecter par la suite dans le programme en cours. De nombreux programmeurs ont un peu été déroutés par l'absence d'équivalent en C. Mais, à l'étape de compilation près, les fonctions `dl*` sont maintenant disponibles pour mimer ce fonctionnement :

```
% cat > dltest.c <<EOF
#include <stdlib.h>
#include <stdio.h>
#include <sys/fcntl.h>
#include <string.h>
#include <dlfcn.h>
int main(void) {
    int fd;
    void *lib;
    void (*funct)();
    char code[] = "#include <stdio.h>\n\
void hello() { printf(\"hello, world\\\n\"); }\n";
    /* Create hello.c */
    if ((fd = open("./hello.c",
                O_WRONLY | O_CREAT, 0640)) < 1) {
        perror("cannot open \"hello.c\"");
        return 1;
    }
    write(fd, code, strlen(code));
    close(fd);
    /* compile hello.c */
    if (
    system("gcc -shared -fPIC -G hello.c -o hello.so") < 0)
        return 2;
    /* read hello.so */
    if (!((lib = dlopen("./hello.so", RTLD_LAZY))))
        return 3;
    /* retrieve "hello" symbol pointer */
    if (!(funct = (void (*)()) dlsym(lib, "hello"))) {
        dlclose(lib);
        return 4;
    }
    /* call funct() */
    funct();
    dlclose(lib);
    return 0;
}
EOF
% gcc dltest.c -o dltest -ldl
```

ATTENTION

Si vous recopiez ce code à la main ou avec un copier/coller, il faut éliminer un des 3 « \ » dans la ligne :

```
void hello() { printf(\"hello, world\\\n\"); }\n";
```

5. Détournements et surcharge

La `libdl` permet aussi de faire des détournements de fonctions.

Admettons que nous voulions afficher un message à chaque `malloc()` et à chaque `free()`. Il nous suffit de réécrire nos deux fonctions et de les faire charger en lieu et place des points d'entrée système par un petit tour de passe-passe. Cependant, il nous faut aussi appeler les fonctions du système afin d'effectuer les opérations d'allocation ou de désallocation.

Le premier tour de passe-passe (le chargement de notre code au lieu du code système) peut se faire de deux façons :

- en recompilant le programme cible et en forçant l'éditeur de liens à utiliser notre bibliothèque ;
- en demandant le préchargement de notre bibliothèque (nettement plus élégant, et *presque* toujours possible).

Concentrons-nous sur la seconde possibilité. Il nous suffit, avant lancement du programme cible, de renseigner la variable `LD_PRELOAD` :

```
# en sh/ksh/bash/zsh ...:
LD_PRELOAD=malib.so programme_cible
# en csh/tcsh:
(setenv LD_PRELOAD malib.so; programme_cible)
```

Prenons un programme qui ne fait que charger une bibliothèque dynamique et attendre :

```
#include <unistd.h>
#include <dlfcn.h>

int main(void) {
    void *p;
    p = dlopen("lib1.so", RTLD_LAZY);
    pause();
    return 0;
}
```

Un programme dépend explicitement de bibliothèques, comme la libc, par exemple. La liste de ces dépendances peut être visualisée grâce à la commande `ldd`.

```
% ldd pause
```

donnera sur Solaris :

```
        libdl.so.1 =>     /usr/lib/libdl.so.1
        libc.so.1 =>      /usr/lib/libc.so.1
        /usr/platform/SUNW,Sun-Fire/lib/libc_psr.so.1
```

En temps normal, dès que le code exécutable d'un programme est chargé en mémoire, les dépendances sont chargées les unes après les autres. Chaque bibliothèque pouvant elle-même dépendre d'autres bibliothèques. Dans le cas d'une bibliothèque chargée par `dlopen()`, ici `lib1.so`, nous avons pendant l'exécution (commande `pldd` sous Solaris) :

```
/usr/lib/libdl.so.1
/usr/lib/libc.so.1
/usr/platform/sun4u/lib/libc_psr.so.1
lib1.so
```

La variable d'environnement indique aux routines de chargement des programmes de charger le contenu de la variable en mémoire juste après le chargement du segment exécutable et juste avant le chargement de la première bibliothèque dans la liste des dépendances.

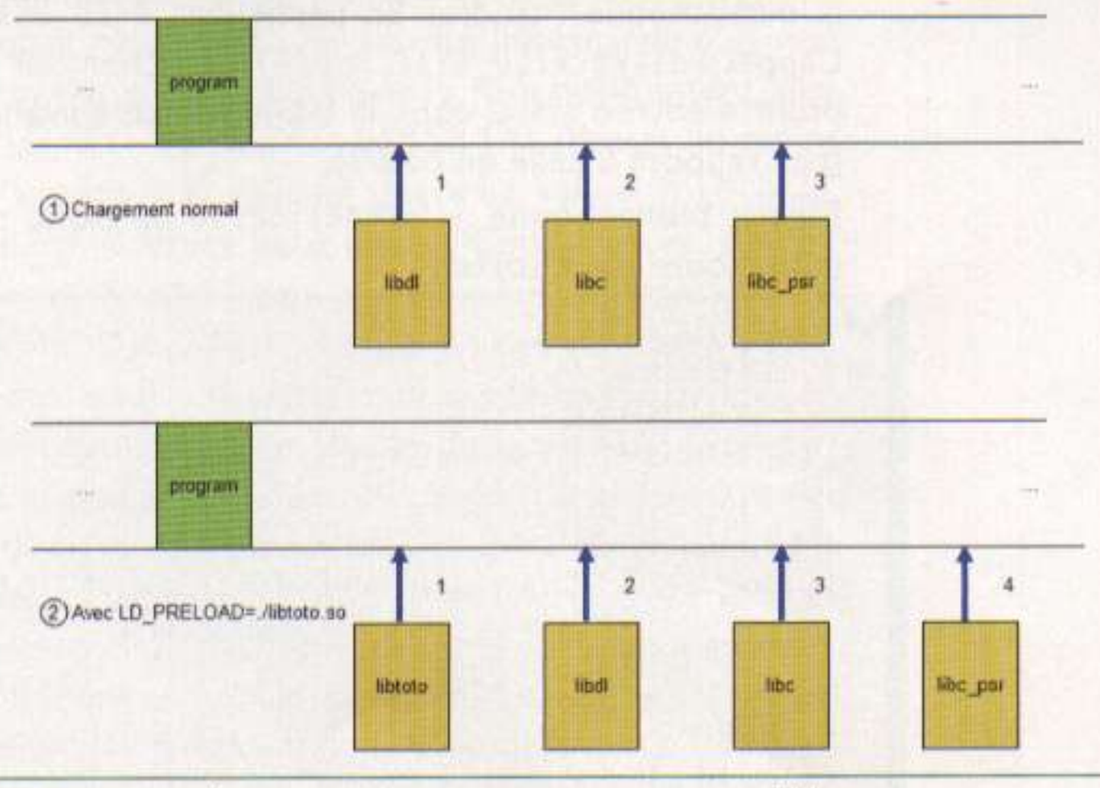
En lançant la commande :

```
% LD_PRELOAD=lib2.so pause
```

On obtient :

```
lib2.so
/usr/lib/libdl.so.1
/usr/lib/libc.so.1
/usr/platform/sun4u/lib/libc_psr.so.1
lib1.so
```

Figure 2 : Chargement normal d'un programme (1) et chargement avec LD_PRELOAD (2)



Supposons maintenant que notre programme cible `tst_alloc` soit le suivant :

```
#include <stdlib.h>

int main() {
    char *str;
    if (!(str = (char *) malloc(32))) return 1;
    free(str);
    return 0;
}
```

que nous compilons de la manière suivante :

```
% gcc -o tst_alloc tst_alloc.c
```

Supposons maintenant le code `mymalloc.c` suivant que nous voulons exécuter :

```
#include <stdio.h>
void * malloc(size_t size) {
    printf("malloc(%d)\n", size);
    return NULL;
}
void free(void *p) {
    printf("free(%x)\n", p);
}
```

que nous compilerons de la façon suivante afin d'en faire une bibliothèque :

```
% gcc -shared -fPIC -o libmymalloc.so mymalloc.c
```

Le lancement de `tst_malloc` seul s'effectue (normalement) sans erreur ni message et retournera 0 (la variable `$?` vaudra 0). Insérons maintenant notre bibliothèque :

```
% LD_PRELOAD=libmymalloc.so tst_malloc
```

affichera :

```
malloc(32)
```

de plus, la commande `echo $?` retournera 1, ce qui est tout à fait normal, puisque notre fonction `malloc` retournera le pointeur 0 et n'effectuera pas l'allocation.

Nous avons surchargé dynamiquement la fonction standard d'allocation mémoire en insérant la nôtre. Voyons maintenant comment appeler le VRAI `malloc` au sein de notre `malloc`. Pour cela, penchons-nous sur l'aide de `dlsym()` (`man dlsym`). Nous voyons que certains *defines* peuvent remplacer le pointeur vers la bibliothèque (handle). En particulier, `RTLD_NEXT`. L'appel à `dlsym(RTLD_NEXT, symbol)` va rechercher le point d'entrée `symbol` dans la bibliothèque suivante (par rapport à celle en cours).

Notre bibliothèque `libmymalloc.so` va donc se compliquer de la sorte :

```
#include <stdio.h>
#include <dlfcn.h>
void * malloc(size_t size) {
    static void *(*sys_malloc)(size_t) = NULL;
    if (!sys_malloc) {
        if (!(sys_malloc =
            (void *(*)(size_t))dlsym(RTLD_NEXT,"malloc"))) {
            perror("cannot fetch system malloc\n");
            exit(1);
        }
    }
    printf("malloc(%d)\n", size);
    return sys_malloc(size);
}
void free(void *p) {
    static void (*sys_free)(void *) = NULL;
    if (!sys_free) {
        if (!(sys_free =
            (void (*)(void *)) dlsym(RTLD_NEXT, "free")))
{
            perror("cannot fetch system free\n");
            exit(2);
        }
    }
    printf("free(%x)\n", p);
    sys_free(p);
}
```

L'étape de compilation de la bibliothèque se voit aussi légèrement modifiée :

```
% gcc -shared -fPIC -G -o \
 libmymalloc.so mymalloc.c -ldl -D_GNU_SOURCE
```

La commande

```
% LD_PRELOAD=libmymalloc.so tst_malloc
```

affichera :

```
malloc(32)
free(20888)
```

(l'adresse en paramètre de `free` est variable en fonction de l'architecture, entre autres) et la variable `$?` devrait maintenant être à 0.

De nombreuses applications de cette technique sont possibles. Elles peuvent aller du débogueur mémoire à l'écriture de chevaux de Troie ou portes dérobées... Certaines restrictions sont cependant tout naturellement imposées pour des questions de sécurité : `LD_PRELOAD` est ignoré par les programmes en suid :).

Le mécanisme de détournement implémente la notion de « surcharge » connue en C++. La `libdl.so` permet aussi d'appeler les fonctions surchargées à l'intérieur des surcharges.

6. Chaînage de fonctions : de l'héritage aux chaînes de traitements

Supposons que nous ayons en mémoire plusieurs bibliothèques qui possèdent des points d'entrée portant le même nom. En généralisant le processus de surcharge, la fonction `dlsym` permet de chaîner ces fonctions.

```
% for i in 1 2 3 4
% do
% cat > lib$i.c <<EOF
lib1.c:
#include <stdio.h>
#include <dlfcn.h>
void foo() {
    void (*next_foo)(void);
    printf("lib$.foo()\n");
    next_foo = dlsym(RTLF_NEXT, "foo");
    next_foo();
}
EOF
% gcc -shared -fPIC -G lib$i.c -o lib$i.so
% done
% cat > foo.c <<EOF
#include <dlfcn.h>
extern void foo();
int main() {
    foo();
}
EOF
% gcc foo.c -o foo -L. -l1 -l2 -l3 -l4 -ldl
```

Pour cet exemple, il est nécessaire que la variable `LD_LIBRARY_PATH` contienne le répertoire courant. Le lancement de `foo` donne :

```
% foo
lib1.foo()
lib2.foo()
lib3.foo()
lib4.foo()
```

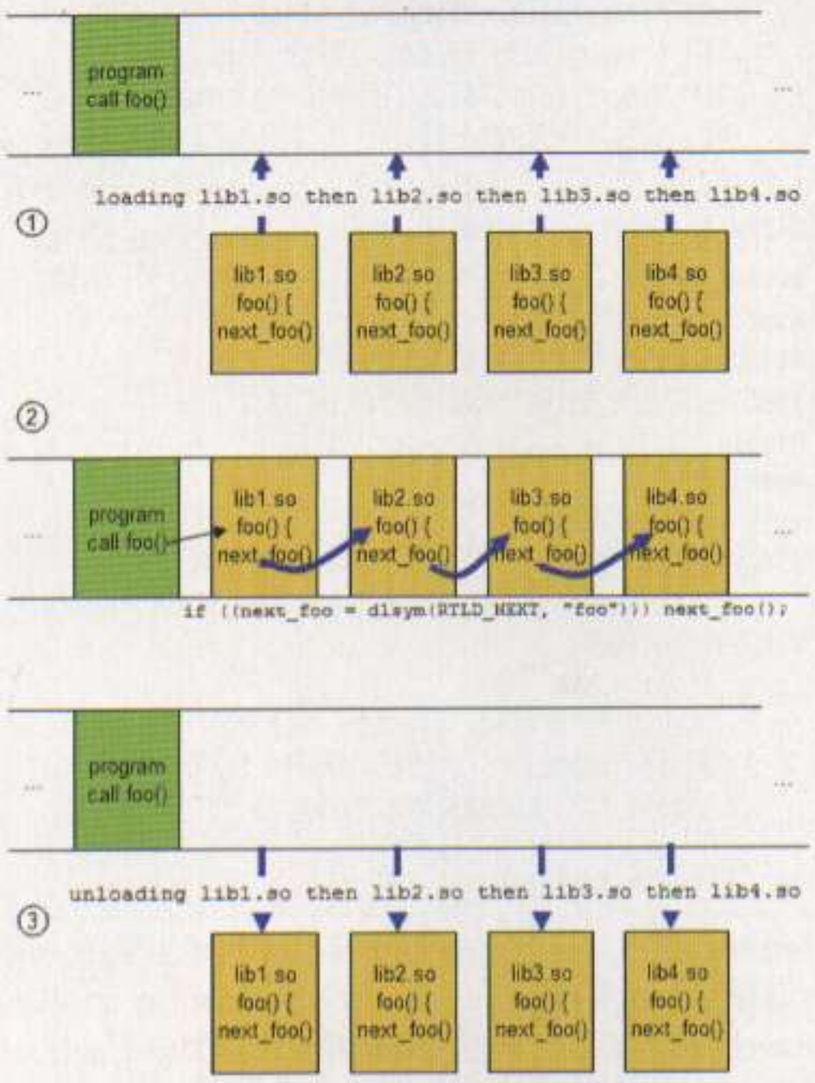
L'ordre d'appel dépend de l'ordre d'inclusion des bibliothèques :

```
% gcc foo.c -o foo -L. -l3 -l2 -l1 -l4 -ldl
% foo
lib3.foo()
lib2.foo()
lib1.foo()
lib4.foo()
```

Voyons s'il en va de même si les bibliothèques sont chargées dynamiquement avec `dlopen`. Il est nécessaire dans ce cas de préciser à l'éditeur de liens à la volée de résoudre tous les points d'entrée dès le chargement et de rendre les symboles globaux (figure 3).

On remplacera dont l'argument `RTLD_LAZY` de `dlopen` par l'argument `RTLD_NOW|RTLD_GLOBAL`.

Figure 3 : Chaînage dynamique de fonctions



(1) Le programme charge dynamiquement les objets partagés `lib1.so`, `lib2.so`, `lib3.so`, puis `lib4.so`.
(2) Le programme appelle la fonction `foo()` déclarée dans la première des bibliothèques chargées. La fonction `foo()` est appelée, effectue son traitement, puis cherche à résoudre le symbole `foo` dans la bibliothèque suivante.
Si ce symbole existe, elle appelle la fonction...
(3) Le programme décharge les bibliothèques.

```
% for i in 1 2 3 4
% cat > libchain$i.c <<EOF
#include <stdio.h>
#include <dlfcn.h>
void foo() {
   void (*next_foo)();
   printf("libchain$i.foo()\n");
   if ((next_foo=(void(*)())dlsym(RTLD_NEXT,"foo")))
        next_foo();
}
EOF
% gcc -shared -fPIC -G libchain$i.c -o libchain$i.so -ldl
% done
% cat > chain.c <<EOF
#include <dlfcn.h>
int main() {
   void *l1, *l2, *l3, *l4;
   void (*foo)();
   l1 = dlopen("libchain1.so", RTLD_NOW | RTLD_GLOBAL);
   l2 = dlopen("libchain2.so", RTLD_NOW | RTLD_GLOBAL);
   l3 = dlopen("libchain3.so", RTLD_NOW | RTLD_GLOBAL);
   l4 = dlopen("libchain4.so", RTLD_NOW | RTLD_GLOBAL);
   foo = (void (*)()) dlsym(l1, "foo");
   foo();
   dlclose(l4);
   dlclose(l3);
   dlclose(l2);
   dlclose(l1);
   return 0;
}
EOF
% gcc chain.c -o chain -ldl
```

Le lancement de `chain` sous Solaris nous donnera :

```
% chain
libchain1.foo()
libchain2.foo()
libchain3.foo()
libchain4.foo()
```

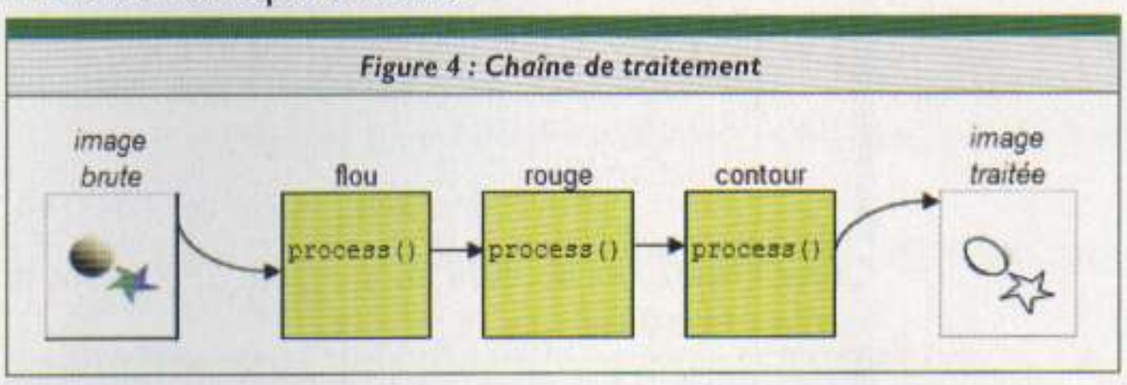
Par contre, sous Linux, `chain` ne nous donnera que :

```
% chain
libchain1.foo()
```

Dans POSIX **[3]**, `RTLD_NEXT` est officiellement « réservé pour une utilisation future » (partie normative). On trouve cependant une section informative (i. e. non normative), qui explique le fonctionnement que `RTLD_NEXT` devra avoir. La LSB **[4]** (depuis au moins la version 1.3) spécifie que « La valeur `RTLD_NEXT`, qui est réservée pour une utilisation future, devrait être disponible, avec le comportement décrit dans ISO POSIX (2003) ». La plupart des pages de manuels sont aussi en adéquation avec la section informative. En conséquence, un bug a été déposé dans le BUGZILLA **[8]** de la glibc. Cependant, eu égard aux positions assez radicales prises par Ulrich Drepper, responsable de la glibc, au sujet de la LSB (« *Do you still think the LSB has some value ?* »), le bug a été, après quelques péripéties, considéré comme invalide par M. Drepper lui-même. Les linuxiens devront donc se rabattre sur la `libtld.so` pour tirer pleinement parti de cette fonctionnalité.

D'ailleurs, que pourrait-on bien en faire ? Le chaînage de fonction permet la création de chaînes de traitement, à l'instar d'une chaîne de montage industrielle. Ce chaînage permet la composition de processus spécialisés. Imaginons, par exemple, que nous fassions du traitement d'image. Une image donnée nécessite une série d'opérations (filtres). Supposons que nous disposions d'une dizaine de filtres spécifiques (flou, rouge, vert, bleu...). Nous pouvons dynamiquement définir un ordre de traitement de notre image de base, puis, par sauts d'une bibliothèque à l'autre.

Figure 4 : Chaîne de traitement



7. Espionnage de greffons

Lorsqu'on programme des greffons, il arrive que ces greffons ne se chargent pas correctement, que certains symboles ne soient pas définis, ou encore que le greffon chargé ne soit pas celui attendu (problème classique du polymorphisme). Il arrive de plus que le programme greffé ne nous transmette pas les messages d'erreur de la `libdl.so`. Dans ces cas, il est assez difficile de trouver les causes des dysfonctionnements.

Pour contourner le problème, on peut détourner les fonctions `dlsym`, `dlopen`, `dlerror` et `dlclose`. Oui, mais voilà, en détournant `dlsym`, comment récupérer le point d'entrée `dlsym` du système ?

Certains systèmes nous facilitent la tâche. Un `nm` de `/usr/lib/libdl.so.1` sur Solaris 8 nous montre que les symboles `dlsym`, `dlopen`, `dlclose` et `dlerror` ont des contreparties en `_dlsym`, `_dlopen`, `_dlclose` et `_dlerror` :

```
nm /usr/lib/libdl.so.1 | egrep "dlsym|dlopen|dlclose|dlerror"
[37]   |  2236|   8|FUNC |GLOB |0    |7      |_dlclose
[35]   |  2244|   8|FUNC |GLOB |0    |7      |_dlerror
[27]   |  2220|   8|FUNC |GLOB |0    |7      |_dlopen
[55]   |  2228|   8|FUNC |GLOB |0    |7      |_dlsym
[26]   |  2236|   8|FUNC |WEAK |0    |7      |dlclose
[53]   |  2244|   8|FUNC |WEAK |0    |7      |dlerror
[47]   |  2220|   8|FUNC |WEAK |0    |7      |dlopen
[44]   |  2228|   8|FUNC |WEAK |0    |7      |dlsym
```

Les points d'entrée standard semblent bien être des *alias* de plus faible priorité que les originaux préfixés par un souligné. Dans ce cas, il est possible d'appeler directement `_dlsym` à l'intérieur de notre fonction détournée.

Sous Linux, par contre, ces alias n'existent pas. Un `nm` de la `libdl.so` ne nous apprendra en fait rien dans les distributions où les bibliothèques système sont débarrassées de leurs symboles (`strip`). L'utilitaire `string` nous donnera les points d'entrée. Las, aucun de ces points d'entrée, après consultation des sources de la glibc (**http://sources.redhat.com/cgi-bin/cvsweb.cgi/libc/dlfcn/?cvsroot=glibc#dirlist**) et quelques tests ne nous sera d'un grand secours.

Il faut alors se tourner vers la fonction `dlvsym(void *handle, const char *name, const char *version_str)` dont la documentation laisse encore à désirer, quelle qu'en soit la source. Cette fonction nécessite, en plus du nom du point d'entrée, une chaîne de caractères représentant la version. Pour retrouver cette chaîne, il suffit d'écrire un petit bout de code faisant appel à `dlsym`, de le compiler et le lier avec la `libdl.so`, puis de faire un `nm` sur le binaire.

```
% nm dltest
...
08049840 W data_start
         U dlclose@@GLIBC_2.0
         U dlopen@@GLIBC_2.1
         U dlsym@@GLIBC_2.0
...
```

Cette chaîne est accolée au symbole `dlsym` sous la forme `dlsym@@version_st` :

```
% nm dltest | grep dlsym | cut -d@ -f3
GLIBC_2.0
```

Une fois cette chaîne trouvée, nous pouvons résoudre le point d'entré, et réécrire nos fonctions :

```
#include <stdio.h>
#include <stdarg.>
#ifdef  _LINUX_
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#endif
#include <dlfcn.h>
```

```
typedef struct {
    void *handle;
    void *(*open)(const char  *, int);
    void *(*sym)(void *, const char *);
    char *(*error)(void);
    int   (*close)(void *);
} dlspy_t;
dlspy_t __dlspy__ = { NULL, NULL, NULL, NULL, NULL };
#ifdef LINUX
#include "config.h"
#ifndef DLSPY_GLIBC_VERSION
#define DLSPY_GLIBC_VERSION "GLIBC_2.0"
#endif
#endif
void
dlspy_init() {
    /* 1st step: save standard dl* functions */
#if defined(SUN)
    __dlspy__.sym   =
      (void *(*)(void *, const char *)) _dlsym;
    __dlspy__.open  =
      (void *(*)(const char *, int))    _dlopen;
    __dlspy__.error =
      (char *(*)(void))                 _dlerror;
    __dlspy__.close =
      (int   (*)(void *))               _dlclose;
#elif defined(LINUX)
    if (!(__dlspy__.sym = dlvsym(RTLD_NEXT,
          "dlsym", DLSPY_GLIBC_VERSION))) {
        perror("dlspy fatal: cannot fetch "
          "system's dlsym entry point\n");
        exit(1);
    }
    __dlspy__.open  = (void *(*)(const char *, int))
        __dlspy__.sym(RTLD_NEXT, "dlopen");
    __dlspy__.error = (char *(*)(void))
        __dlspy__.sym(RTLD_NEXT, "dlerror");
    __dlspy__.close = (int   (*)(void *))
        __dlspy__.sym(RTLD_NEXT, "dlclose");
#else
#error ERROR: cannot get dl* original entry points.;
#endif
}
void
dlspy_echo(char *fmt, ...) {
    va_list args;
    char buf[1024], buf2[1024];
    va_start(args, fmt);
    sprintf(buf2, "dlspy: %s", fmt);
    vsprintf(buf, buf2, args);
    fprintf(stderr, "%s\n", buf);
    va_end(args);
}
char *

dlerror(void) {
    static char *errstr = NULL;
    char *t;
    if (!__dlspy__.open) dlspy_init();
    t = __dlspy__.error();
    if (!t && errstr) return errstr;
    if (t) {
        errstr = t;
        return t;
    }
    return (char *) strdup(" ");
}
void *
dlopen(const char *name, int mode) {
    void *t = NULL;

    if (!__dlspy__.open) dlspy_init();
```

```
    dlspy_echo("try to open \"%s\" shared object"
      " with mode %d...", name, mode);
    if (!(t = __dlspy__.open(name, mode)))
      dlspy_echo("failed (%s)\n", dlerror());
    else dlspy_echo("ok\n");
    return t;
}
void *
dlsym(void *h, const char *name) {
    void *t = NULL;
    if (!__dlspy__.open) dlspy_init();
    dlspy_echo("try to bind symbol \"%s\" "
      "as new entry point...", name);
    if (!(t = __dlspy__.sym(h, name)))
      dlspy_echo("failed (%s)\n", dlerror());
    else dlspy_echo("ok at %x\n", t);
    return t;
}
int
dlclose(void *h) {
    if (!__dlspy__.open) dlspy_init();
    dlspy_echo("dlclose(%p)", h);
    return __dlspy__.close(h);
}
```

Peut-on aller plus loin ? Une fonctionnalité intéressante serait de pouvoir tracer chaque appel aux fonctions résolues par `dlsym()`. Pour ce faire, il semble nécessaire de définir un pool de fonctions qui devront, lors du premier appel, prendre l'adresse du point d'entrée fraîchement résolu, et, dans un second temps, appeler le point d'entrée avec les paramètres qui lui ont été passés, sans pour autant connaître leur nombre ni leur taille.

Ce problème n'est pas nouveau. La FAQ du groupe de discussion comp.lang.c **[9]** en parle comme d'un problème insoluble de manière générique. Des pointeurs ou axes de recherches y sont explorés. On peut le résoudre en grande partie par l'utilisation de certaines fonctionnalités obscures de gcc **[10]**, les `__builtin_apply` et `__builtin_apply_args`.

8. Et en C++ ?

Il est possible d'utiliser la `libdl.so` en C++. Cependant, il est nécessaire de faire un peu plus attention qu'en C. En effet, l'implémentation du polymorphisme (mais qui pourrait tout autant servir à vérifier le typage en cours d'exécution) en C++ passe par l'ajout de préfixes et de suffixes aux noms de fonction. C'est la technique du *mangling*. Aussi, est-il nécessaire, comme pour l'utilisation de pointeurs de fonctions, de définir les fonctions exportées en « `extern "C"` » afin d'éviter le mangling.

Pour aller plus loin sur le sujet, voir les documents cités en référence **[11]** et **[12]**.

Conclusion

Cette présentation de la `libdl` est loin d'être exhaustive. Les possibilités qu'elle offre, comme la surcharge ou l'héritage à un niveau très différent d'un langage de programmation en font un outil très puissant dont la seule limite est celle notre propre imagination.

Yann Langlais,

langlais@ilay.org

RÉFÉRENCES

- **[1]** *Executable and Linkable Format :* **http://en.wikipedia.org/wiki/Executable_and_Linkable_Format**
  Tool Intrerface Standard (TIS) : Executable and Linkable Format (ELF) Specification v1.2 : **http://refspecs.freestandards.org/elf/elf.pdf**
  Tool Intrerface Standard (TIS) : Generic ELF Specification ELFVERSION : **http://www.linuxbase.org/spec/book/ELF-generic/ELF-generic.html**
  LEVINE (John), *Linkers and Loaders, Chapter 10 : « Dynamic Linking and Loading »*, **http://www.iecc.com/linker/linker10.html**
- **[2]** *The GNU Libtool Homepage :* **http://www.gnu.org/software/libtool/** **http://www.gnu.org/software/libtool/manual.html**
- **[3]** *The Open Group Base Specifications Issue 6 IEEE Std 1003.1*, 2004 Edition (i. e. la norme POSIX)
  Single Unix Specification : **http://www.unix.org/single_unix_specification/**
  Voir les spécifications de dlopen : **http://www.opengroup.org/onlinepubs/009695399/functions/dlsym.html**
  Ce site demande un enregistrement gratuit préalable.
- **[4]** *Linux Standard Base 3.0*
  13.16. Interface Definitions for libdl : **http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/libdlman.html**
  13.15. Data Definitions for libdl : **http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/libdl-ddefs.html**
  dlsym : **http://refspecs.freestandards.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/baselib-dlsym-1.html**
- **[5]** *Linux Programer's Manual :* **http://www.frech.ch/man/man3/dlopen.3.html**
- **[6]** *Linux HOWTO :* **http://www.faqs.org/docs/Linux-HOWTO/Program-Library-HOWTO.html**
  How to Write Shared Libraries : http://people.redhat.com/~drepper/dsohowto.pdf
- **[7]** L'éditeur de lien et ses variables d'environnement : DRALET (Samuel), *GNU/Linux Magazine France* numéro 63, juillet/août 2004, page 76.
- **[8]** Descriptif et historique du bug 1319 sur le comportement de dlsym avec RTLD_NEXT : **http://sourceware.org/bugzilla/show_bug.cgi?id=1319**
- **[9]** *comp.lang.c Frequently Asked Questions : 15. Variable-Length Argument Lists :* **http://www.eskimo.com/~scs/C-faq/s15.html**
- **[10]** *gcc info : Construction calls :* **http://www.cs.cmu.edu/cgi-bin/info2www?(gcc.info)Constructing%2520Calls**
  gcc features : WOLF (Clifford), « *Some demonstrations of nice/osbcure gcc features* », **http://www.clifford.at/cfun/gccfeat/**
- **[11]** *C++ dlopen mini HOWTO :* **http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO/C++-dlopen-mini-HOWTO.html**
- **[12]** NORTON (James), « *Dynamic Class Loading for C++ on Linux* », *Linux Journal*, **http://www.linuxjournal.com/article/3687**
  LANGLAIS (Yann) : **http://ilay.org/yann/articles/**

Yves Bailly

Le langage Ada : un peu d'assembleur

Le langage Ada est un langage généraliste de haut niveau. Mais l'une de ses vocations premières était l'informatique embarquée, au plus proche du matériel. Comme tous les langages offrant la possibilité de « toucher à l'électronique », Ada permet d'intégrer directement du code assembleur au sein d'un programme plus vaste.

Naturellement, la manière d'intégrer l'assembleur ainsi que le contenu même de ce code dépendent fortement de la plate-forme visée, aussi bien de l'architecture matérielle que du compilateur utilisé. Ce que nous allons voir ici concerne les machines à base de processeur de la famille 80x86, sous environnement GNU/Linux, en utilisant le compilateur Ada GNAT tel qu'il se trouve intégré à la suite GCC.

Certains s'interrogent peut-être sur la pertinence d'utiliser l'assembleur, surtout de nos jours où le moindre morceau de puce peut être manipulé par une API en langage C, généralement fournie par le fabricant. On pourrait alors se contenter d'interfacer l'API C en Ada, comme nous l'avons vu précédemment (voir *Linux Magazine* 85).

Mais, il est des situations où l'assembleur peut s'avérer indispensable. Il serait (théoriquement) parfaitement possible d'écrire le pilote d'une carte vidéo en Ada, bénéficiant ainsi de toute la puissance et robustesse du langage – mais dans des cas de ce genre, les petites latences induites par une couche d'interface avec les fonctions C sont tout simplement inacceptables, car dégradant les performances. Plus généralement, dans une situation où la vitesse d'exécution est primordiale (imaginez la simulation d'un modèle météorologique ou astronomique), le recours à l'assembleur peut permettre d'optimiser très fortement certaines portions du programme bien plus efficacement qu'un compilateur – nous verrons un exemple dans cet article.

Enfin, imaginez vouloir écrire un système d'exploitation en Ada. Le recours à l'assembleur est incontournable à certains endroits clefs – même un système écrit en C comme Linux n'y échappe pas. Et si vous pensez qu'écrire un système en Ada est une idée farfelue et irréaliste pour autre chose qu'une sonde martienne, jetez un œil au projet *Toy Lovelace* **[1]**.

Intégrer l'assembleur

Commençons par l'intégration d'une instruction assembleur au sein d'une procédure Ada (dans un programme `rien.adb`) :

```
with System.Machine_Code;
use System.Machine_Code;
procedure Rien is
begin
   Asm("nop");
end Rien;
```

L'assembleur est intégré au code Ada au moyen de la procédure `Asm()`, déclarée dans le paquetage `System.Machine_Code`. Le premier paramètre de `Asm()` est une chaîne de caractères contenant les instructions à exécuter. Ici, nous invoquons l'instruction `nop`, synonyme de *no operation*, c'est-à-dire très précisément « ne rien faire ».

L'assembleur étant ce qu'il est, il peut s'avérer fort utile de pouvoir consulter le code généré pour l'ensemble du programme, afin de voir précisément comment les instructions que nous écrivons sont intégrées au reste – surtout quand cela ne fonctionne pas. Pour cela, exécutez simplement :

```
$ gcc -c -S -gnatp rien.adb
```

Ceci va générer un fichier `rien.s` contenant le code assembleur équivalent à l'ensemble du programme. Les paramètres passés à `gcc` sont :

- `-c` pour seulement compiler, sans chercher à effectuer d'édition des liens ;
- `-S` pour générer, justement, le fichier `rien.s` ;
- `-gnatp` désactive (presque) tous les mécanismes de contrôle internes au langage Ada (vérification des intervalles de valeurs, ce genre de choses), ce qui allège le code généré.

Sur notre exemple, on obtient ceci :

```
        .file   "rien.adb"
        .text
.globl _ada_rien
        .type   _ada_rien, @function
_ada_rien:
.LFB3:
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
#APP
        nop
#NO_APP
        popl    %ebp
        ret
.LFE3:
        .size   _ada_rien, .-_ada_rien
...
```

Plus encore un certain nombre de lignes propres à l'utilisation du *runtime* Ada par Gcc. Ce qui nous intéresse commence à la ligne `_ada_rien` : on retrouve le nom de notre procédure principale, préfixé par `_ada_`. La plupart des instructions ne concernent que la gestion de la pile (elles n'apparaîtraient pas si nous avions passé l'option `-fomit-frame-pointer` à `gcc`).

Finalement, notre code assembleur se trouve encadré par les lignes `#APP` et `#NO_APP` : il est donc facilement repérable – du moins dans un petit programme.
Mais passons à quelque chose de plus intéressant.

Entrées et sorties

L'instruction assembleur `div` effectue une division entière, en retournant le quotient et le reste. Normalement, pour obtenir ces deux valeurs, il est nécessaire d'effectuer deux opérations. Écrivons une procédure qui invoque `div` pour les obtenir directement :

```
1  with Ada.Text_IO;
2  with System.Machine_Code;
3  with Interfaces;
4  procedure Div is
5     use ASCII;
6     use Interfaces;
7     use System.Machine_Code;
8     use Ada.Text_IO;
9     procedure Div(dividende: in     Integer;
10                  diviseur : in     Integer;
11                 quotient :    out Integer;
12                 reste    :    out Integer) is
13       quotient_interne: Integer_32 := 0;
14       reste_interne   : Integer_32 := 0;
15    begin
16       Asm("movl $0, %%edx " & LF & HT &
17           "div %%ebx "      & LF & HT &
18           "movl %%eax, %0 " & LF & HT &
19           "movl %%edx, %1 ",
20           Inputs => (Integer_32'Asm_Input("a",
             Integer_32(dividende)),
21                      Integer_32'Asm_Input("b",
                        Integer_32(diviseur))),
22           Outputs => (Integer_32'Asm_Output("=m",
             quotient_interne),
23                       Integer_32'Asm_Output("=m",
                         reste_interne)),
24           Clobber => ("edx"));
25       quotient := Integer(quotient_interne);
26       reste    := Integer(reste_interne);
27    end Div;
28...
29 -- etc.
```

Voilà qui est sensiblement plus compliqué. Voyons cela étape par étape.

Pour commencer, le type `Integer_32` introduit lignes 13 et 14 représente un entier signé codé sur 32 bits. Ce type est déclaré dans le paquetage `Interfaces`. On s'assure ainsi de parfaitement maîtriser le format des données que l'on va échanger avec l'assembleur. En effet, le type standard `Integer` est simplement défini comme étant « un type entier ». Sa taille (voire sa représentation) peut varier selon la plate-forme, aussi est-il indispensable d'utiliser un type parfaitement délimité quand on manipule l'assembleur, qui supporte assez mal l'approximation dans ce domaine.

Ensuite, vous constatez que la chaîne de caractères contenant l'assembleur est littéralement construite morceau par morceau (lignes 16 à 19), en insérant ce curieux `& LF & HT` entre deux lignes. Ces deux symboles, déclarés de type `Character` dans le paquetage `ASCII`, représentent les caractères ASCII « retour chariot » (*Line Feed*) et « tabulation horizontale » (*Horizontal Tab*). Ils sont introduits afin d'améliorer la présentation du code assembleur généré. Si nous avions écrit simplement ceci :

```
Asm("movl $0, %%edx " &
    "div %%ebx "      &
    "movl %%eax, %0 " &
    "movl %%edx, %1 ",
```

Voici ce que nous aurions obtenu en examinant le code source assembleur obtenu par `gcc -S` :

```
...
        movl    32(%esp), %ebx
#APP
        movl $0, %edx div %ebx movl %eax, 4(%esp) movl %edx, (%esp)
#NO_APP
        movl    4(%esp), %eax
...
```

Ce qui est, convenez-en, parfaitement illisible – et généralement refusé par l'assembleur GNU. L'ajout de ces deux caractères donne un résultat bien plus sympathique et passe-partout :

```
...
        movl    32(%esp), %ebx
#APP
        movl $0, %edx
        div %ebx
        movl %eax, 4(%esp)
        movl %edx, (%esp)
#NO_APP
        movl    4(%esp), %eax
...
```

Les entrées

Les entrées sont données par le paramètres `Inputs` de la procédure `Asm()` (lignes 20 et 21). Ce paramètre attend une valeur, ou un tableau de valeurs, d'un type retourné par l'attribut `'Asm_Input`. Cet attribut, tout comme le contenu du paquetage `System.Machine_Code`, est dépendant de l'implémentation – ici, nous ne traitons que du compilateur GNAT, basé sur GCC. Répétons-le, un compilateur différent utilisera des moyens (types, procédures et attributs) différents pour obtenir un effet équivalent.

Le premier paramètre de l'attribut est une chaîne de caractères donnant une *contrainte* (*constraint*) sur la valeur que l'on souhaite passer à l'assembleur, laquelle valeur est donnée en deuxième paramètre. On retrouve là les éléments qui sont communiqués à l'instruction `asm()` de GCC. Pour reprendre un exemple de la documentation de GNAT, l'instruction suivante dans un code en C :

```
asm("fsinx %1 %0" : "=f" (resultat) : "f" (angle));
```

...se présenterait ainsi en Ada (du moins pour le compilateur GNAT) :

```
Asm("fsinx %1 %0",
    Outputs => Float'Asm_Output ("=f", resultat),
    Inputs  => Float'Asm_Input  ("f",  angle));
```

C'est plus long à écrire, mais cela semble plus lisible. Naturellement, le type de la valeur donnée en deuxième paramètre doit être le même que le type utilisé en préfixe de l'attribut :Ada est un langage au typage strict. C'est pourquoi nous effectuons ici un transtypage, `dividende` et `diviseur` étant de type `Integer`, alors que nous voulons un type `Integer_32`. On obtient ainsi un minimum de vérifications, donc de robustesse.

Nous ne nous étendrons pas sur le contenu de la chaîne de caractères. Il correspond à la syntaxe propre à GCC (voir le chapitre « *Extensions to the C Language Family/Assembler Instructions with C Expression Operands* » et le suivant dans la documentation de GCC). Disons simplement qu'ici nous demandons à ce que la valeur (transtypée) de `dividende` soit placée dans le registre `EAX`, tandis que la valeur (transtypée) de `diviseur` est placée dans le registre `EBX`. Le compilateur se chargera lui-même de ces actions.

Les sorties

Vous l'aurez sans doute deviné, les sorties sont données par le paramètre `Outputs` de la procédure `Asm()`, en utilisant l'attribut `'Asm_Ouput` (lignes 22 et 23). Le principe général est le même que pour les entrées, sauf que le deuxième paramètre de l'attribut doit être une variable et non une simple valeur. C'est pourquoi il a été nécessaire de déclarer deux variables temporaires, `quotient_interne` et `reste_interne`, du type `Integer_32`.

Pour information, la contrainte `"=m"` indique que l'on désigne une sortie (le signe `=`) qui sera référencée par une adresse mémoire (la lettre `m`). En pratique, cela signifie que les écritures `%0` et `%1` dans le code assembleur seront remplacées par les adresses effectives des variables `quotient_interne` et `reste_interne`.

Intégrer gentiment

Enfin, notre code assembleur commence par placer la valeur 0 dans le registre `EDX` (ligne 16), car l'instruction `div` (dans notre contexte) suppose que la valeur à diviser est contenue dans la paire `EDX:EAX`. De plus, `EDX` est également utilisé pour contenir le reste de la division. Seulement, le registre `EDX` n'est référencé ni dans les paramètres d'entrées, ni dans les paramètres de sortie. Le compilateur ne peut donc pas « deviner » que le registre `EDX` est utilisé et modifié par notre code assembleur, car il n'analyse pas celui-ci. Cela n'a pas grande conséquence ici, mais cela pourrait être catastrophique dans un programme plus vaste : le compilateur pourrait parfaitement utiliser ce registre `EDX` pour ses besoins propres, notre code perturbant alors le déroulement du programme.

Aussi est-il généralement nécessaire de passer un quatrième paramètre à la procédure `Asm()`, nommé `Clobber` (ligne 24). Celui-ci attend une chaîne de caractères contenant les noms des registres utilisés par le code assembleur, séparés par une virgule. Par exemple, `"eax, ebx, ecx"`. Si nécessaire, le compilateur s'adaptera et le programme pourra se dérouler normalement.

Amélioration... ou pas

Généralement, les programmeurs qui utilisent l'assembleur ont le souci constant de l'optimisation. Le code de l'exemple précédent pourrait être rédigé ainsi :

```
Asm("cltd "       & LF & HT &
    "div %%ebx ",
    Inputs => (Integer_32'Asm_Input("a", Integer_32(dividende)),
               Integer_32'Asm_Input("b", Integer_32(diviseur))),
    Outputs => (Integer_32'Asm_Output("=a", quotient_interne),
                Integer_32'Asm_Output("=d", reste_interne)));
```

C'est-à-dire qu'on réduit au minimum les instructions insérées. Le changement essentiel se situe sur les paramètres de sortie : plutôt que d'indiquer une adresse mémoire, on nomme explicitement les registres remplis par l'instruction `div`. Par ailleurs, le paramètre `Clobber` a disparu, devenu inutile (car `EDX` est mentionné dans la deuxième contrainte).

Le tableau suivant compare la partie « intéressante » de l'assembleur généré, à gauche la première version, à droite la version « améliorée » :

```
    movl    $0, 4(%esp)
    movl    $0, (%esp)
    movl    28(%esp), %eax
    movl    32(%esp), %ebx
#APP
    movl $0, %edx
    div %ebx
    movl %eax, 4(%esp)
    movl %edx, (%esp)
#NO_APP
```

```
    movl    $0, 8(%esp)
    movl    $0, 12(%esp)
    movl    28(%esp), %eax
    movl    32(%esp), %ebx
#APP
    cltd
    div %ebx
#NO_APP
    movl    %eax, 8(%esp)
    movl    %edx, 12(%esp)
```

Mis à part l'utilisation de `cltd` pour mettre à 0 le registre `EDX`, ces deux codes sont en fait équivalents. Peut-être peut-on considérer qu'une version est plus « parlante » que l'autre, mais l'autre est moins « dangereuse » que l'une...

Calculs en virgule flottante

Si vous pratiquez le développement d'applications graphiques en 3D, il est probable que vous ayez fréquemment à calculer le sinus et le cosinus d'un angle donné. La plupart du temps, les deux sont d'ailleurs nécessaires dès que l'on manipule quelque chose ressemblant de près ou de loin à un cercle. Or, il existe une instruction assembleur qui permet justement de calculer ces deux valeurs en une seule fois : `fsincos`. Voilà qui semble un bon candidat si on se trouve dans un objectif d'optimisation.

Réalisons donc un petit programme pour vérifier que l'utilisation de l'assembleur apporte effectivement un gain de temps :

```
with Ada.Text_IO;              use Ada.Text_IO;
with Ada.Real_Time;            use Ada.Real_Time ;
with Ada.Numerics;             use Ada.Numerics;
with Interfaces;               use Interfaces;
with System.Machine_Code;      use System.Machine_Code;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;
procedure Sin_Cos is

   procedure Sin_Cos(angle: in     Long_Float;
                     sin_a:    out Long_Float;
                     cos_a:    out Long_Float) is
begin
   sin_a := Sin(angle);
   cos_a := Cos(angle);
end Sin_Cos;
pragma Inline(Sin_Cos);
```

Les fonctions sinus et cosinus pour les flottants en double précisions (`Long_Float` en Ada) sont déclarées dans le paquetage `Ada.Numerics.Long_Elementary_Functions`, aux côtés de nombreuses autres fonctions élémentaires. Pour commencer, on crée une procédure classique réalisant ce que nous voulons, par des moyens « normaux ». Remarquez la directive `Inline` qui suit la procédure, elle informe le compilateur d'étendre le code correspondant « en ligne » afin d'économiser un appel de fonction.

Voyons maintenant la version assembleur :

```
procedure Sin_Cos_Asm(angle: in     Long_Float;
                      sin_a:    out Long_Float;
                      cos_a:    out Long_Float) is
begin
   Asm("fsincos",
     Inputs  => Long_Float'Asm_Input("0", angle),
     Outputs => (Long_Float'Asm_Output("=t", cos_a),
                 Long_Float'Asm_Output("=u", sin_a))
      );
end Sin_Cos_Asm;
pragma Inline(Sin_Cos_Asm);
```

Non, pas d'erreur : le code assembleur se résume au seul appel de `fsincos`. En fait, tout le travail pénible est effectué par les déclaration des entrées/sorties. L'instruction opère sur la valeur sur la pile du co-processeur (`ST0`) en la dépilant, puis empile le sinus *puis* le cosinus – on récupère donc les résultats dans `ST1` et `ST0`, respectivement. Ces résultats sont obtenus par les contraintes `"u"` et `"t"` (respectivement), comme toujours précédées sur signe `"="` signalant qu'il s'agit de valeurs de retour.

Mais comme `fsincos` attend également une valeur en `ST0`, lequel registre est également utilisé pour le résultat, on passe la contrainte `"0"` pour la déclaration de la valeur d'entrée, signalant ainsi qu'elle doit être placée dans le même emplacement que la première valeur de retour.

De fait, toutes les manipulations nécessaires de la pile du co-processeur sont automatiquement assurées par le compilateur, ce que l'on peut vérifier en jetant un œil à l'assembleur généré :

```
$ gcc -c -S -gnatp sin_cos.adb && cat sin_cos.s
...
sin_cos__sin_cos_asm.394:
.LFB5:
        pushl   %ebp
.LCFI12:
        movl    %esp, %ebp
.LCFI13:
        subl    $24, %esp
.LCFI14:
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        movl    %eax, -24(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -20(%ebp)
        fldl    -24(%ebp)
#APP
        fsincos
#NO_APP
        fstpl   -8(%ebp)
        fstpl   -16(%ebp)
        fldl    -16(%ebp)
        fstpl   (%edx)
        fldl    -8(%ebp)
        fstpl   8(%edx)
        movl    %edx, %eax
        leave
        ret     $4
...
```

Naturellement, il convient tout de même de vérifier que l'on ne s'est pas trompé quelque part, par exemple en intervertissant les résultats.

Vous aurez sans doute remarqué que nous avons utilisé directement le type `Long_Float`. Cela fonctionne, car, dans notre situation précise (compilateur GNAT, machine 80x86), ce type correspond exactement aux nombres à virgule flottante sur 64 bits du processeur. Si toutefois vous vous placez sur une architecture différente, avec un compilateur différent, il peut s'avérer nécessaire d'effectuer un transtypage comme nous l'avons fait précédemment. Pour information, le paquetage Interfaces qui accompagne le compilateur GNAT comporte les types `IEEE_Float_32` et `IEEE_Float_64` pour les nombres à virgule flottante sur 32 et 64 bits respectant la norme IEEE.

Côté performances, le calcul de dix millions de sinus/cosinus par la méthode classique prend environ 5.86 secondes sur la machine de votre serviteur, tandis qu'en utilisant l'assembleur cela prend... moins d'une seconde.

Passer des adresses mémoires

Pour finir, voyons comment nous pouvons passer l'adresse de blocs de données depuis l'Ada vers l'assembleur. L'exemple consiste à reproduire une fonctionnalité de la procédure `Put_Line()`, permettant d'afficher une chaîne de caractères. L'affichage sera en fait réalisé par la fonction système `write()`. Enfin, nous pourrons saluer le monde comme il se doit.

Une adresse mémoire est représentée en Ada par le type `Address`, déclaré dans le paquetage `System`, l'adresse d'un objet étant fournie par l'attribut `'Address`. Contrairement au langage C, il ne s'agit pas (forcément) d'un type équivalent à un entier. C'est généralement le cas, en particulier sur plate-formes x86, mais pas forcément. Pour un maximum de portabilité, il est donc nécessaire de convertir une valeur de type `Address` en une valeur d'un type entier reconnaissable directement par l'assembleur.

Le paquetage `System.Storage_Elements` contient justement une fonction `To_Integer()` effectuant cela, retournant une valeur de type `Integer_Address` (déclaré dans `System.Storage_Elements`). Comme pour le type standard `Integer`, la taille de ce type n'est pas réellement connue, car dépendante de la plate-forme, mais, au moins, nous savons qu'il s'agit d'un entier que nous pourrons transtyper vers, par exemple, `Integer_32`.

Le programme commence donc par définir une fonction `To_Int_32()` recevant une adresse mémoire et retournant un `Integer_32` :

```
with System.Machine_Code;
with Interfaces;
with System.Storage_Elements;
procedure Put_Line is
   use ASCII;
   use Interfaces;
   use System.Machine_Code;
   function To_Int_32(addr: in System.Address)
      return Unsigned_32 is
   begin
      return Unsigned_32(System.Storage_
Elements.To_Integer(addr));
   end To_Int_32;
```

Remarquez que contrairement à ce que nous faisons d'habitude, le paquetage `Ada.Text_IO` n'est pas référencé. Voici notre version de `Put_Line()` :

```
   procedure Put_Line(s: in String) is
      new_s: constant String := s & LF;
   begin
      Asm("movl  $1,%%ebx " & LF & HT &
          "movl  $4,%%eax " & LF & HT &
          "int   $0x80 ",
          Inputs => (Unsigned_32'Asm_Input("d",
new_s'Length),
                     Unsigned_32'Asm_Input("c",
To_Int_32(new_s(1)'Address))),
          Clobber => ("eax, ebx")
         );
   end Put_Line;
   pragma Inline(Put_Line);
```

Il n'y a en fait rien de bien particulier par rapport à ce que nous avons déjà vu. La procédure commence par créer une nouvelle chaîne à partir de la première, en lui ajoutant le caractère `LF` (pour le retour à la ligne). Les entrées du code assembleur consistent, dans l'ordre :

- à placer la longueur de la chaîne dans le registre `EDX` ;
- à placer l'adresse du premier caractère de la chaîne dans le registre `ECX`.

C'est là qu'intervient notre fonction `To_Int_32`. La seule petite subtilité consiste à prendre l'adresse du *premier caractère* de la chaîne (par `s(1)'Address`), plutôt que l'adresse de la chaîne elle-même (par `s'Address`). En effet, en Ada, une chaîne est un tableau de caractères, comme en C, mais contrairement au C, un tableau n'est pas qu'une adresse mémoire. Il est donc possible qu'en mémoire les données d'un tableau soient précédées d'informations « administratives », comme la taille du tableau, ses bornes, etc. Prendre directement l'adresse de la chaîne pourrait donc aboutir à l'affichage d'octets indésirables. On évite cela en prenant explicitement l'adresse du premier caractère. Naturellement, si la chaîne est vide, invoquer `s(1)` provoquera une exception : un code véritable devrait donc s'assurer qu'il y a au moins un caractère à adresser.

Enfin, on place dans `EBX` l'identifiant du descripteur de fichier voulu (ici 1, correspondant à la sortie standard `stdout`), puis dans `EAX` l'identifiant de la fonction système voulue (ici 4, la fonction `write()`). La dernière instruction déclenche l'interruption 0x80, utilisée sous Linux pour invoquer les fonctions système du noyau.

On termine le programme par le message consacré :

```
begin
   Put_Line("Bonjour, Monde !");
end Put_Line;
```

Résultat :

```
$ gnatmake put_line.adb && ./put_line
Bonjour, Monde !
$
```

Conclusion

Voilà pour cette petite présentation de l'utilisation de l'assembleur en Ada, du moins dans le cadre du compilateur GNAT utilisé sur une architecture 80x86. Ceci montre bien qu'Ada est un langage véritablement généraliste pouvant être utilisé dans toutes les situations, même les plus exigeantes.

Yves Bailly,

http://www.kafka-fr.net

RÉFÉRENCES

- **[1]** Toy Lovelace : **http://ipnnarval.in2p3.fr/~xavier/**
- **[2]** Codes sources de l'article : **http://www.kafka-fr.net/articles/presentation-ada05_17-sources.tar.bz2**