GNU LINUX MAGAZINE / FRANCE

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

L 19275 - 93 - F: 6,20 €

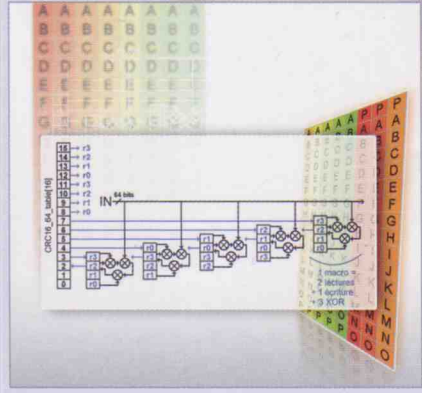
▶AVRIL ▶2007 ▶NUMÉRO 93

Développez vos extensions Firefox

p. 65

Comprenez l'architecture XUL/JavaScript/XPCOM/Gecko, puis apprenez à développer une extension pour enfin la distribuer sous la forme d'un paquet XPI.

Algorithmique, C et optimisation



Signature ultrarapide de blocs de données

Étudiez le fonctionnement d'un algorithme de signature, puis de son implémentation. Enfin, apprenez à optimiser vos sources tout en conservant leur portabilité.

p. 49

Administration et développement sur systèmes UNIX

V.D.S.

- *Un système dédié complet sans aucune limitation. (Distribution Debian accès root intégral - panel de gestion ultra simplifié).*
- *Des ressources réservées et confortables (Ram, CPU, I/O, Bande Passante).*
- *Un prix bas pour un rapport qualité/prix unique.*

Bande passante incluse de 2 Mbps à 6 Mbps en trafic illimité - Espace disque de 4 à 40 Go - mémoire vive de 64 Mo à 512 Mo

à partir de

9 €

HT/mois

www.sivit.fr

V.D.S.

Serveur Dédié Virtuel

Sivit

Edito

Vers du matériel plus ouvert...

Voilà le prochain défi qui s'ouvre aux Logiciels libres. Nous, utilisateurs de ces technologies, sommes en mesure de maîtriser et d'auditer (ou faire auditer) le code ouvert des systèmes, des serveurs et des applications. Mais cela n'est pas suffisant. Pour obtenir la maîtrise complète de son infrastructure informatique, qu'elle soit celle d'un particulier, d'une entreprise, d'une multinationale ou d'un gouvernement, il faut pouvoir connaître et comprendre tous les éléments qui la composent.

Une (micro-)étape vient d'être franchie dernièrement avec la mise en production d'une nouvelle carte mère supportée par LinuxBIOS (et non d'une carte mère Gigabyte M57SLI-S4 pré-équipée avec un BIOS ouvert). La logique intégrée dans les cartes mères et les périphériques est aujourd'hui aussi riche, puissante et complexe que les ordinateurs d'antan. Le code propriétaire que ce matériel contient répond aux mêmes problématiques que les systèmes d'exploitations : faille de sécurité, problème de stabilité, etc.

La solution est pourtant simple et nous savons comment améliorer la qualité d'un logiciel. Il suffit d'ouvrir les sources et de distribuer le code sous les termes d'une licence permettant à chacun d'utiliser, de comprendre et d'améliorer ce code. Les fabricants ou, du moins, une partie d'entre eux commencent à prendre en compte les arguments en faveur de l'ouverture des sources. Cependant, il s'agit d'un monde bien différent de celui de l'édition de logiciels. Les mœurs ne sont pas les mêmes et l'inertie importante. On parle d'industrie du matériel où la notion de start-up innovatrice n'existe pas vraiment. Certains gros acteurs sont encore tentés de répondre que leurs pilotes ou que les caractéristiques précises de leurs produits relèvent du secret industriel. Diffuser ces informations revient, pour eux, à aider leurs concurrents.

Voilà pourquoi la FSF et d'autres groupes défendant la cause du Logiciel libre tiennent à mettre l'accent sur le matériel et à pousser les constructeurs à dépasser leurs craintes. Le Logiciel libre a prouvé sa valeur dans le software. Il est maintenant temps que le hardware en profite également.

Sur ces belles paroles, je vous donne rendez-vous au 28 avril pour le prochain numéro. D'ici là, remplissez-vous la tête de toutes les bonnes choses qui peuplent les pages qui suivent...

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
Secrétaire de rédation:
Véronique Wilhelm
Conception graphique :
Fabrice Krachenfels
Responsable publicité :
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

Debian Etch, « French translation completed »

Le support des langues est un point clef de toute distribution GNU/Linux. L'utilisateur lambda oublie souvent que les caractéristiques multilingues des distributions sont quelque chose d'exceptionnel et nécessitent un travail important de la part des contributeurs.

Ce mois-ci, j'ai donc décidé de mettre en avant le travail de personnes comme Christian Perrier et toute l'équipe française de traduction. En effet, dans un *post* début mars, Christian a informé la communauté des développeurs Debian que l'ensemble des paquets de la branche Unstable disposaient de leurs éléments debconf francisés.

Les quelques 70 contributeurs à cet effort colossal ont travaillé pendant presque 5 ans pour arriver à ce résultat. Bien entendu, cela a été rendu possible également grâce à la participation des responsables de paquets ayant remonté les bogues de traduction et aidé à la correction.

C'est peut-être un peu chauvin, mais il n'en reste pas moins très agréable de voir que la traduction française est au top de l'avancement des travaux avec, juste derrière, le tchèque, l'allemand, le suédois et le vietnamien. On remarquera avec amusement que le classement en question (http://www.debian.org/intl/l10n/po-debconf/rank) n'a pas grand-chose à voir avec le classement des langues les plus parlées au monde.

On notera également que les équipes de traduction pour les différentes langues ne font pas que mettre du cœur à l'ouvrage. La réactivité est également au rendez-vous. Ainsi, début janvier, un appel fut lancé sur les listes de diffusion afin de corriger au plus vite les bogues de traduction. Dans le même temps, une campagne NMU fut lancée. Un NMU pour « *Non-Maintainer Uploads* » est un processus permettant à une personne qui n'est pas le responsable d'un paquet de mettre à jour celui-ci.

Typiquement, les NMU sont intimement liés aux traductions, puisque le responsable d'un paquet n'est pas en mesure de traduire vers toutes les langues supportées par la distribution (ni même une seule dans la plupart des cas). Après la campagne, de nombreux bogues restaient ouverts et tout le monde a remonté ses manches pour finalement arriver au résultat qui est celui que nous connaissons.

Ce petit Debian Corner est donc l'occasion de remercier l'équipe de traduction française et de saluer le travail accompli. Merci ! :)

La prochaine étape sera d'atteindre les 100% de traduction pour Testing. Pour l'heure, nous en sommes à 99.2 %. L'objectif n'est pas si loin, mais l'aide est toujours la bienvenue. Si vous souhaitez participer à l'effort de traduction, inscrivez-vous aux listes de diffusion adéquates et consultez http://www.debian.org/intl/l10n/po-debconf/fr. Pour plus de renseignements sur les traductions en rapport avec debconf, rendez-vous sur http://www.debian.org/international/french/po-debconf.

La traduction n'est pas qu'une affaire d'interface de configuration (debconf), il faut également s'occuper des fichiers `.po` (gettext), des applications elle-mêmes et des autres éléments multilingues. Je pense naturellement aux paquets comme `openoffice.org-l10n-fr` ou `gimp-help-fr`. Ce n'est pas nécessairement des membres des équipes de traduction qui sont en charge de ces paquets, mais cela fait partie du support.

Enfin, il reste un élément et non des moindres : l'installeur. Bien entendu, un utilisateur normal (au sens GLMF du terme) ne se frottera à l'interface d'installation que très rarement. On ne réinstalle pas une distribution Debian, on la met à jour et on la reconfigure. Mais l'installeur est la première chose que voit un utilisateur de sa distribution. La traduction est donc un élément clef pour le convaincre qu'il a fait le bon choix. Et là encore, le français est totalement et parfaitement supporté !

davfs2

Montage WebDave

Voici un paquet très intéressant. Non pas tant pour son utilité, monter une ressource WebDAV (berk !) comme un système de fichiers, mais pour sa conception. `davfs2` permet la gestion d'un système de fichiers depuis l'espace utilisateur... sans utiliser Fuse.

`davfs2` utilise uservfs (anciennement appelé PODFUK pour « *POrtable Dodgy Filesystems in Userland (hacK)* »). Cette solution, quasi concurrente de Fuse, ne nécessite pas de patch noyau ou de module complémentaire. C'est une fonctionnalité présente dès les versions 2.2.X de Linux qui est utilisée : le support codafs. Au final, `davfs2` vous servira surtout de base d'étude pour les interactions noyau/utilisateur en ce qui concerne les systèmes de fichiers pour éventuellement développer autour d'uservfs.

`davfs2` utilise la `libneon2` pour le protocole WebDAV et supporte l'utilisation de SSL et d'un serveur mandataire.

Le BTS contient quelques entrées pour ce paquet. Il s'agit en majorité de points concernant les traductions debconf (très courants dans le BTS en ce moment).

Denis Bodor,

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

Éric Lacombe, Matthieu Barthélemy

Syslets/Threadlets et recherches sur la gestion de pile résau

Ce mois-ci, nous espérons encore ravir les amateurs de techniques internes au noyau en vous proposant ces deux brèves. Le premier sujet dissèque et synthétise le travail proposé par Ingo Molnar, destiné à apporter la gestion des appels système asynchrones dans le noyau, et montre comment en tirer parti dans les applications. Le second se penche sur la couche réseau et expose une nouvelle vision de celle-ci, imaginée par des chercheurs et des développeurs. Nous nous retrouvons le mois prochain pour la synthèse de la vingt-et-unième mouture de la série 2.6 de notre noyau, qui marquera la dixième édition du Kernel Corner. Saisissant l'occasion offerte par ce chiffre rond, n'hésitez pas à nous soumettre vos opinions sur la rubrique via nos adresses mail !

[NOUVELLE FONCTIONNALITÉ] SYSLETS/THREADLETS OU L'EXÉCUTION ASYNCHRONE DES APPELS SYSTÈME

Éric Lacombe – tuxiko@free.fr, eric.lacombe@security-labs.org

Introduction

Suite aux discussions sur la façon d'ajouter à Linux le support de l'exécution asynchrone pour les appels système, Ingo Molnar a développé au courant du mois de février, une approche nommée « *syslet* ». La première version publiée définit le concept d'atome de syslet : objet élémentaire représentant un appel système à traiter. Les *benchmarks* effectués révèlent alors la pertinence de l'approche face aux entrées/sorties synchrones (que ce soit dans le cas où le cache est désactivé ou l'inverse !). La version suivante (la v3 ;), entraîne des modifications majeures au niveau de l'ABI et apporte également une nouvelle fonctionnalité : les *threadlets*, couche de haut niveau reposant sur les syslets. La version 4 ne contient qu'une modification de l'API des threadlets. La dernière version publiée à ce jour (la v5) apporte principalement le support de l'architecture x86_64.

L'infrastructure des syslets

Avant d'expliquer le fonctionnement des syslets, prenons l'exemple d'une application souhaitant lire tous les fichiers situés dans un répertoire. En utilisant les appels système classiques, lorsqu'elle accède à ces fichiers, il est fort possible qu'ils ne soient pas présents en cache, auquel cas la fonction `read` va bloquer (ou renvoie une erreur si l'appel est configuré comme non bloquant) jusqu'à ce que le VFS (*Virtual File System*) ait récupéré les données sur le périphérique de stockage. Il s'agit ici d'un comportement synchrone. C'est l'approche employée pour la quasi-totalité des appels système sous Linux.

L'approche asynchrone donne à l'application l'opportunité de soumettre un traitement au noyau sans bloquer et d'être averti de sa terminaison. Pour cela, l'infrastructure des syslets définit un composant de base appelé « atome ». Il s'agit d'une enveloppe contenant un appel système qui est ensuite soumise au noyau pour qu'il soit exécuté. La soumission se fait au travers de l'appel système `sys_async_exec`, lequel prend en paramètre un atome.

Voyons à quoi ressemblent ces atomes. Chacun est représenté par une structure contenant le numéro de service noyau à appeler ainsi que ses paramètres. Un membre est également réservé pour la valeur retournée à l'issue de l'exécution. D'autres champs figurent dans cette structure dont le membre `next` et `flags` servant respectivement à chaîner les atomes et à contrôler leur flux d'exécution. Ainsi, bien que l'appel système précédent ne prenne qu'un seul atome en paramètre (l'atome initial), plusieurs pourront être exécutés.

Une application, utilisant ces atomes pour communiquer au noyau les services qu'elle requiert, va profiter d'un traitement optimal de ses requêtes. En effet, après la soumission d'un appel système via un atome, le noyau va l'exécuter. S'il ne bloque pas, le comportement est identique au cas synchrone : la fonction retourne à l'espace utilisateur. Par contre, s'il bloque, c'est-à-dire que les données ne sont pas disponibles (*cachemiss*), le noyau va appeler `schedule` (c.-à-d. l'ordonnanceur). À cet instant, le sous-système des syslets va récupérer d'une réserve, un *cachemiss thread* lequel va servir de médium à l'exécution de la suite de l'application alors que le thread actuel exécutera l'appel système bloquant. Ainsi, elle pourra effectuer d'autres traitements en attendant la terminaison de l'appel système.

Nous l'avons mentionné, le chaînage des atomes permet à l'application d'envoyer en un tout plusieurs

[NOUVELLE FONCTIONNALITÉ] SYSLETS/THREADLETS OU L'EXÉCUTION ASYNCHRONE DES APPELS SYSTÈME

Éric Lacombe – tuxiko@free.fr, eric.lacombe@security-labs.org

appels système. Leur flux d'exécution est conditionné par les flags de chaque atome. Ainsi, il est possible de créer de petits « programmes » (ou scripts) s'exécutant en espace noyau (sans avoir à retourner en espace utilisateur avant la fin de l'exécution).

Reprenons notre exemple. Nous souhaitons lire l'ensemble des fichiers dans un répertoire. En utilisant les syslets, nous avons seulement à créer un atome avec le flag `SYSLET_STOP_ON_NON_POSITIVE`. Ainsi, la fonction `read` est exécutée en boucle dans le noyau jusqu'à ce que tout le fichier soit lu. Nous pouvons aussi ajouter le flag `SYSLET_SKIP_TO_NEXT_ON_STOP` et ainsi déclencher l'exécution de l'atome suivant lorsque tout le fichier sera lu (sans pour autant sortir du contexte du noyau). L'atome suivant (chaîné via le membre `next`) est dans notre cas l'appel système `close`. A la fin de la lecture d'un fichier, ce dernier est alors automatiquement fermé. Notons que lors de l'exécution de ces atomes, plusieurs cachemiss thread pourront être utilisés (par exemple lors des différentes lectures).

Finissons avec une brève explication sur le fonctionnement de la notification. En effet, l'utilisateur a soumis des appels système au noyau, mais comment prend-t-il connaissance de la fin de leur exécution ? De même, comment récupère-t-il les résultats ? Pour cela, il initialise en espace utilisateur, via la fonction `async_head_init`, une structure `struct async_head_user` contenant principalement un tampon mémoire circulaire. Lorsqu'il soumet ensuite les appels système qu'il souhaite effectuer via `sys_async_exec`, il passe également la structure précédente en paramètre. Lorsque le noyau termine l'exécution d'un appel système, il inscrit dans la première entrée libre du tampon, l'adresse de l'atome correspondant. À la charge ensuite de l'utilisateur de libérer les entrées (en leur affectant la constante `NULL`) après en avoir pris connaissance, car le noyau n'écrira pas de nouvelles valeurs si le tampon est rempli. Il renverra à la place un code d'erreur.

Afin d'être alerté quant à la terminaison d'un ou plusieurs atomes, l'utilisateur peut se mettre en attente sur la fonction `sys_async_wait` qui prend en paramètre le nombre minimum d'atomes devant s'être terminés avant de retourner.

Vous l'avez certainement remarqué, l'utilisation des syslets est astucieuse et peut révéler quelques complexités de mise en œuvre. C'est le prix à payer pour un outil efficace et offrant de nombreuses possibilités. Néanmoins, l'utilisation de cette infrastructure est plutôt destinée aux bibliothèques se devant de fournir à leurs utilisateurs un service performant. Ainsi, une couche de plus haut niveau a été ajoutée et met à la disposition des applications une interface plus accessible. C'est ce que nous allons voir dans la prochaine partie.

La couche des Threadlets

Les threadlets ont été conçues à la suite des critiques sur la complexité d'utilisation des syslets. Ainsi, les threadlets représentent une sur-couche à l'infrastructure des syslets, dans laquelle le concept des atomes est masqué à l'utilisateur. L'utilisation directe des syslets reste pertinente dans les cas où nous souhaitons ne faire aucun compromis entre la facilité d'utilisation et les performances.

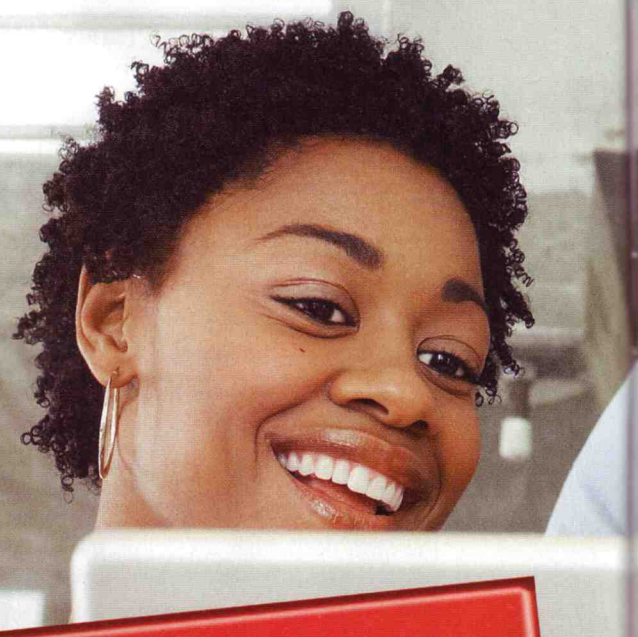
Pour exécuter nos appels système de façon asynchrone, il suffit dorénavant d'écrire dans une fonction le code dont l'exécution est appelée à être asynchrone. Cette fonction est alors nommée « threadlet ». Elle doit respecter un prototype particulier et se terminer par la fonction `threadlet_complete`. Nous l'exécutons ensuite via la fonction `threadlet_exec`.

L'application profite donc d'un parallélisme étant établi à la volée en fonction du besoin (c.-à-d. des blocages éventuels des appels système). Ce dernier est déterminé de façon optimale au sein du noyau. Avant de continuer dans l'explication, illustrons cela par un bout de code issu d'un exemple donné par Ingo Molnar :

```
long my_threadlet_fn(void *data)
{
        char *name = data;
        int fd;
        fd = open(name, O_RDONLY);
        if (fd < 0)
                goto out;
        fstat(fd, &stat);
        read(fd, buf, count)
        ...
out:
        return threadlet_complete();
}
main()
{
        done = threadlet_exec(threadlet_fn, new_stack, &user_head);
        if (!done)
                reqs_queued++;
}
```

La fonction `threadlet_exec` prend en argument la threadlet, mais également deux autres paramètres. `new_stack` est un pointeur sur une zone mémoire qui servira, si besoin, comme pile pour les threads créés lorsqu'un ou plusieurs appels système bloquent dans la threadlet. Le dernier argument `user_head` est

1&1, tout pour votre site Web



Votre succès en 3 étapes :

- **Réalisez facilement votre site Web,**
  grâce à d'innombrables outils de création
- **Passionnez et fidélisez vos visiteurs,**
  grâce aux flux d'infos en temps réel
- **Faites parler de vous,**
  grâce aux solutions interactives 1&1

Avec 1&1, votre avenir sur le Web est entre de bonnes mains et votre site, vraiment prêt à suivre toutes vos envies.

Nouveau :

L'ACTUALITÉ EN TEMPS RÉEL

avec 1&1 Contenu Dynamique

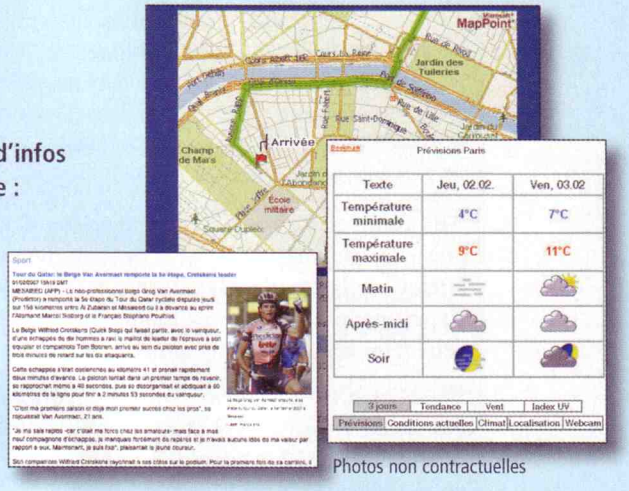
Transformez votre site en mine d'infos actualisées en temps réel !

1&1 vous propose une solution pour intégrer des flux d'informations en temps réel, jusqu'alors réservée aux plus grands sites…

Grâce aux offres de contenu dynamique présentées ici, vous avez la possibilité d'intégrer des informations réactualisées automatiquement au fil de la journée sur votre site. En effet, il vous suffit de choisir le ou les contenus qui vous intéressent et une fois mis en place, vous n'avez plus rien à faire. 1&1 s'occupe de la mise à jour ! Et ce n'est pas tout, cette solution vraiment innovante est totalement gratuite !

Découvrez ici quelques-uns des flux d'infos que vous pouvez intégrer à votre site :

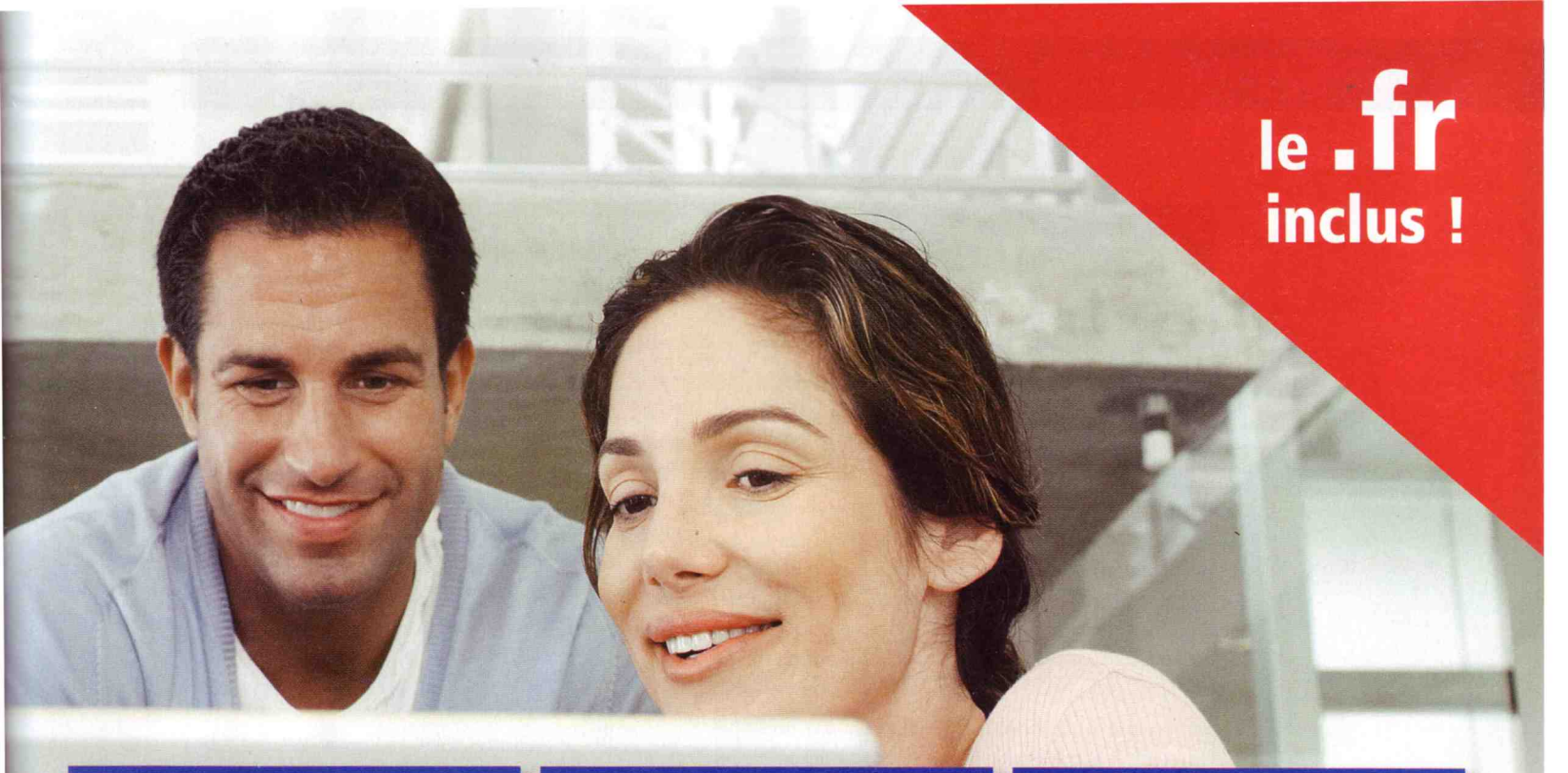
- La Une de l'actualité
- France
- Économie / Finance / Bourse
- High Tech
- Sports / Football
- Culture / Art de vivre
- Nouvelles « People »
- Itinéraires…



Photos non contractuelles

MEMBRE DE united internet

N° INDIGO **0 825 080 020** (0,15 € TTC la minute)



le .fr inclus !

PACK PERSO INITIAL

0,99 € HT/mois
1,18 € TTC/mois

1 domaine en .fr, .com, .net, .org, .info

Pour les particuliers exigeants qui souhaitent se lancer dans la création d'un site Web sans aucune connaissance en programmation.

- 1500 Mo d'espace
- 25 Go de trafic
- 10 comptes email
- 1&1 Blog
- 1&1 Contenu Dynamique

Et bien plus encore...

PACK PERSO CONFORT

4,99 € HT/mois
5,97 € TTC/mois

2 domaines en .fr, .com, .net, .org, .info

Pour les associations et petits commerçants qui désirent disposer d'une vitrine attrayante sur le Net et bénéficier de nombreuses solutions clé en main.

- 6000 Mo d'espace
- 750 Go de trafic
- 200 comptes email
- 5 bases de données MySQL
- 1&1 Contenu Dynamique

Et bien plus encore...

PACK PRO STANDARD

9,99 € HT/mois
11,95 € TTC/mois

3 domaines en .fr, .com, .net, .org, .info

Pour les petites et moyennes entreprises qui ont besoin d'un site Web dynamique et interactif pour renforcer leur activité.

- 10 000 Mo d'espace
- 1000 Go de trafic
- 1200 comptes email
- 20 bases de données MySQL
- 1&1 Contenu Dynamique

Et bien plus encore...

Windows News CHOIX DE LA RÉDACTION

« Très pro et très accessible »
Windows News, Février 2007, n°153

1&1

www.1and1.fr

[NOUVELLE FONCTIONNALITÉ] SYSLETS/THREADLETS OU L'EXÉCUTION ASYNCHRONE DES APPELS SYSTÈME

Éric Lacombe – tuxiko@free.fr, eric.lacombe@security-labs.org

l'adresse de la structure dont nous avons déjà parlé, contenant le tampon circulaire pour la notification de la terminaison des appels asynchrones.

Cette fonction retourne la valeur 1 dans le cas où la threadlet s'est déroulée sans blocage. Par contre, si un appel système bloque, un nouveau thread est créé pour poursuivre l'exécution de la threadlet. Le thread original attend, quant à lui, la terminaison de l'appel système. Notons que la gestion des signaux reste correcte dans ce nouveau thread si des gestionnaires avaient été mis en place (`CLONE_SIGNAL` et `CLONE_SIGHAND` sont utilisés).

Précisons que `threadlet_exec` n'est pas un appel système. Cette fonction fait appel à `sys_threadlet_on`, véritable appel système qui va demander au noyau de passer en « mode d'exécution asynchrone », en bref, d'utiliser les syslets.

De même, la fonction `threadlet_complete` qui termine la threadlet, exécute au final l'appel système `sys_threadlet_off` qui renvoie 1 dans le cas où l'exécution se passe dans le contexte de l'application, sinon 0. Il s'agit alors d'une exécution dans un thread asynchrone. Auquel cas, la fonction `async_thread` est appelée afin : (1) d'enregistrer dans le tampon circulaire de l'application, l'évènement relatif au traitement asynchrone qui vient de se terminer, et (2) de replacer le thread dans la réserve utilisée pour les exécutions asynchrones.

Conclusion

Nous avons omis plusieurs détails par souci de simplicité. Nous vous laissons le loisir de parcourir les sources du projets à l'adresse **http://redhat.com/~mingo/syslet-patches/** pour compléter votre formation ;) Notez cependant que ce sous-système est toujours en cours d'évolution. Ce n'est que lorsqu'il sera intégré à la *mainline*, signifiant son entière maturité, qu'il conviendra de l'utiliser.

[RECHERCHE] PILE RÉSEAU, NETWORK CHANNELS ET GRAND UNIFIED FLOW CACHE

Matthieu Barthélemy – bonsouere@gmail.com

Nous faisons à nouveau un tour dans le monde de la recherche gravitant autour du noyau Linux, et nous penchons cette fois sur la pile réseau. Extrêmement complète sous Linux, elle intègre bien sur la gestion des paquets réseau, mais aussi leur filtrage, classification et surveillance via la solution Netfilter/IPTables ou encore la flexibilité des algorithmes utilisés si l'on pense aux choix possibles dans les protocoles de congestion TCP. Elle n'est pas limitée aux classiques TCP, IP et UDP, mais implémente aussi des protocoles tels que UDP-lite, DCCP, SCTP... Témoignage de la réputation de notre noyau, on le retrouve ainsi dans des équipements professionnels spécifiquement dédiés au réseau : *firewalls*, routeurs, répartiteurs de charge... Sa gestion du réseau est un des éléments clefs de son succès, mais elle n'est pas exempte de problèmes ou de pistes d'améliorations. Plusieurs personnes se sont intéressées de près à son fonctionnement et à ses problèmes, et des solutions ont été discutées pour améliorer son organisation et ses performances.

I – Fonctionnement et problèmes soulevés

La couche réseau est chargée d'assurer l'acheminement des données entre un pilote d'interface réseau et le destinataire final de ces données (application...). Lors d'une communication réseau, les données se décomposent en paquets. Historiquement, le fonctionnement de Linux voulait qu'à chaque paquet reçu par l'interface réseau physique (carte Ethernet...), une interruption matérielle soit générée envers le gestionnaire d'interruptions du pilote de cette interface. Nous n'expliquerons pas pourquoi c'est une solution coûteuse ; sachons seulement que dans les faits ceci est très consommateur en ressources au fur et à mesure que la charge en débit augmente. Linux 2.6, puis Linux >= 2.4.20, introduisent NAPI (New API), qui apporte à ce niveau une nouvelle manière de traiter les données entrantes. Désormais, pour les pilotes implémentant NAPI, une telle interruption matérielle est générée beaucoup moins fréquemment grâce au *polling*. Lors d'une première « salve » de paquets, la carte génère effectivement l'interruption ; le pilote en prend connaissance et désactive aussitôt les interruptions pour être « tranquille ». En contrepartie, le pilote implémente une méthode `poll()` qui est appelée, et qui, pendant n itérations, attend les paquets, décrémentant n à chacun d'entre eux. Une fois n revenu à zéro, les paquets sont traités par la couche réseau, et le pilote accepte à nouveau les interruptions. NAPI a été largement adoptée par les

[RECHERCHE] PILE RÉSEAU, NETWORK CHANNELS ET GRAND UNIFIED FLOW CACHE

Matthieu Barthélemy – bonsouere@gmail.com

pilotes, ses bénéfices étant prouvés. Notons cependant, car ceci n'est pas innocent, qu'un seul processeur à la fois peut appeler `poll()`.

La couche réseau de Linux stocke les informations sur les paquets et leur destination (*routing*) dans plusieurs structures de type tables de hachage. Ces structures existantes, visibles dans les fichiers sources `include/net/inet_hashtables.h` et `net/ipv4/inet_hashtables.c`, sont communes à toute la pile réseau, et leur code contient de nombreux verrous destinés à respecter l'atomicité des opérations sur système multiprocesseur.

Début 2006, le développeur Van Jacobson, connu pour ses travaux d'amélioration des protocoles TCP/IP, lance le débat sur la pertinence de l'organisation de la couche réseau dans les systèmes Unix lors d'une présentation orale. Dérivé du père d'Unix, MULTICS, le concept de base n'a pas changé depuis 1980 :

- Une pile entièrement en espace noyau : la pile est donc globale et commune à toutes les parties l'utilisant. Sous trop forte charge, le noyau est configuré pour ignorer purement et simplement les paquets qu'il n'a pas le temps de traiter, impactant ainsi tous les services connectés sans distinction. Les applications orientées temps réel s'en trouvent particulièrement pénalisées. Cette pile en espace noyau effectue également des compromis pénalisants dus à sa généricité.
- Un cheminement coûteux : réception d'un paquet par l'interface physique -> interruption matérielle -> réception par le pilote -> interruption logicielle -> `skb` (*socket buffer*), classification globale du paquet par le noyau (est-il entrant, sortant, est-il lu par son destinataire ou pas ?) -> passage par le « chemin lent » : Netfilter et/ou routage -> attribution éventuelle à un socket -> `read()` éventuel par l'application. Plusieurs interruptions soft sont en fait nécessaires au traitement d'un paquet. À chaque étape, changement de contexte, et chaque traitement n'est pas forcément effectué sur le même processeur, ce qui induit de coûteuses opérations de transfert d'informations entre cache (mémoire) et processeur(s). Des améliorations ont été apportées pour attribuer le plus d'étapes possibles du traitement au même processeur, mais une affinité complète n'est pas encore possible.

Vous suivez toujours ? Parfait. Chaque paquet reçu est mis dans une structure, elle-même insérée dans une table de hachage ; ainsi, au niveau TCP, on en distingue trois :

- `tcp_bhash` (sockets *bound*, disposant d'une adresse locale) ;
- `tcp_ehash` (sockets *connected*);
- `tcp_listening_hash` (*listening*, sockets en écoute).

La liste des sockets existants sur un système peut être obtenue via `/proc/net/tcp`, qui parcourt ces tables. Faisons un test simpliste et grossier. Sur un système bi-Xeon (soit 4 processeurs vus) avec 1600 sockets utilisés :

```
$time cat /proc/net/tcp > /dev/null

real    0m0.520s
user    0m0.000s
sys     0m0.452s
```

Le même système, avec 20 sockets utilisés :

```
$time cat /proc/net/tcp > /dev/null

real    0m0.014s
user    0m0.000s
sys     0m0.008s
```

Sans aucune prétention quant à la rigueur de ce test, on voit ici que le temps de traverse des tables de hachage, appelées très fréquemment par le code réseau, peut devenir conséquent. Lors du parcours brut d'une telle table, sous Linux, un verrou (`read_lock()`) est posé à la lecture de chaque élément.

Entre le moment où un paquet est réceptionné par le pilote et celui où il est délivré à l'application en espace utilisateur, le paquet est classifié par le pilote et placé dans une structure au noyau ; si Netfilter est activé, chaque paquet passe dans sa moulinette afin de déterminer s'il correspond à une ou des règles de filtrage mises en place. C'est un gros problème pour Van Jacobson, qu'il appelle *slow path*.

II – Une solution originale

La présentation de Van Jacobson, les performances ainsi que la réduction du coût en ressources processeur qu'il annonce lancent le débat et l'intérêt autour du concept de « *Network Channels* » (ou *Netchannels*). Un Netchannel est un lien direct entre un producteur (le pilote d'interface réseau) et un consommateur (l'application en espace utilisateur) : le flux de données va directement de l'un à l'autre, sans couche(s) intermédiaire(s) en espace noyau. Evgeniy Polyakov, auteur de l'infrastructure Kevent (cf. KC92), développe et prône également l'utilisation de ces canaux réseau. Au niveau implémentation,

[RECHERCHE] PILE RÉSEAU, NETWORK CHANNELS ET GRAND UNIFIED FLOW CACHE

Matthieu Barthélemy – bonsouere@gmail.com

un canal est principalement une suite de structures paquets (`sk_buff`). Si Van Jacobson n'a jamais publié aucun code attestant ses dires et fut critiqué sur ce point, les travaux de Polyakov sont publiquement disponibles et régulièrement améliorés dans l'espoir d'aboutir à une implémentation complète. Après une période enthousiaste ayant succédé à la présentation de Van Jacobson et au premier patch de Polyakov, les arguments en faveur des Netchannels ont été démontés un par un par les développeurs sceptiques. Evgeniy Polyakov s'est employé à contre-argumenter longuement, benchmarks à l'appui, pour prouver sa vision, sans que rien n'ait pu réellement être tranché en faveur des uns ou des autres. Parmi les objections soulevées, la latence pouvant être introduite par une exploitation *userspace* des couches TCP/IP, qui pourrait entraîner un renvoi de paquets de confirmation (paquets `ACK`) trop tardif. Malgré tout, plusieurs développeurs noyau demeurent très intéressés par cette approche. Notons que Kelly Daly (IBM) et Rusty Russell ont également proposé leur implémentation basée sur les travaux de Van Jacobson.

Actuellement, Polyakov en est à sa vingtième version des netchannels. Du côté noyau, son code assure le suivi de connexion (*connection tracking*), la NAT (translation d'adresse) et bien sûr le *binding* et l'implémentation des canaux. Son implémentation en espace utilisateur gère la réception TCP et UDP, le marquage temporel des paquets, le routage, une interface fournissant des sockets au milieu applicatif...

Dans les travaux des deux développeurs, une seule étape s'immisce entre le pilote et le canal : il s'agit de déterminer à quel canal est destiné un paquet. Vous aurez peut-être remarqué que nous n'avons pas parlé de protocole, seulement de transport de flux de données... Effectivement, le modèle des Netchannels laisse le soin à l'espace utilisateur d'implémenter la majeure partie de la couche TCP/IP. Le principe est : décaler l'interprétation du flux le plus proche possible du consommateur, selon le principe que tout le monde n'a pas besoin de la même quantité ou de la même nature de traitements. Polyakov fournit une implémentation en espace utilisateur de pile TCP/IP, encore rudimentaire.

Pour creuser le sujet, le site de Polyakov :
http://tservice.net.ru/~s0mbre/old/?section=projects&item=netchanne

III – Le Grand Unified Flow Cache

Van Jacobson soutient que les structures actuelles, utilisées pour classifier un paquet au sein du noyau, sont une charge inutile. Evgeniy Polyakov va jusqu'à assumer que l'implémentation actuelle du routage dans le noyau n'est pas nécessaire, et est partisan d'une structure unique pouvant supplanter les TCP *hashtables*, qui serait plus performante, y compris lors de grosses charges en paquets, et surtout ne nécessitant pas de verrous. Rusty Russel, auteur de Netfilter/IPtables, appuie l'idée et imagine une structure appelée *Grand Unified Flow Cache* (GUFC). Dans la vision des deux développeurs, elle se chargerait uniquement de déterminer le channel à emprunter par un paquet, et le pousserait dans celui correspondant. Le GUFC serait une structure permettant d'identifier de manière unique chaque paquet et sa destination, selon des *tuples* qui pourraient être organisés ainsi :

adresse source – port source – adresse de destination – port de destination – protocole – source locale ou non – destination locale ou non

Si les caractéristiques d'un paquet entrant correspondent à un tuple du GUFC associé à un Netchannel, le paquet y est immédiatement envoyé, sans passer par aucun autre filtrage ; ceci pourrait être un moyen de court-circuiter le passage de certains flux par Netfilter.

Si plusieurs développeurs de la couche réseau Linux s'accordent sur l'idée, l'implémentation est encore matière à débats enflammés (confinant au troll) sur les algorithmes tels que tables de hachage vs listes vs arbres binaires. Polyakov essaie depuis longtemps de démontrer la supériorité de sa structure en arbre, appelée « trie », qu'il affirme être exempte de tout besoin de verrouillage.

Dans leur forme actuelle, les Netchannels ne s'accordent pas forcément bien avec le concept de « socket », mais davantage avec une nouvelle solution d'interfaçage avec les applications, asynchrone ou événementielle (Kevent ?).

Pour conclure sur le sujet des performances de la couche réseau, revenons à l'actualité du noyau.

Dans un futur plus proche, une amélioration qui devrait voir le jour (pour Linux 2.6.22 ou 2.6.23 ?) est la possibilité pour plusieurs threads noyau de lire des paquets depuis une interface réseau simultanément, depuis des processeurs différents. Le travail serait effectué au niveau de NAPI, qui deviendrait alors « multiqueue NAPI ». Enfin, la discussion autour d'une infrastructure permettant d'accéder de manière asynchrone à des ressources telles que fichiers ou sockets bat son plein, entre les Kevents de Polyakov et les threadlets d'Ingo Molnar. Dans certains cas, les performances et la réactivité en bénéficient.

La LSM song

La rédaction a récemment reçu par e-mail les paroles d'une chanson, qui parodie « Nathalie », de Oldelaf et Monsieur D. L'auteur a tenu à rester anonyme, prétextant ne pas avoir été inspiré(e) que par la chanson originale (« Malgré l'imagination, il y a des choses qui ne s'inventent pas »). Séduits par le style nerd-folk à tendance résolument débianique, nous n'avons pas hésité à insérer cet interlude musical dans nos colonnes, agrémenté des illustrations de Louna.

On s'est rencontré un dimanche
Lors de l'install party (ouuuh-ouuuh)
Dans la salle polyvalente
Au fond de la mairie
Ton pécé était tout gruiké
Et ne démarrait plus (yéé-éé)
Y'avait un conflit de glibc,
Et l'USB avait fondu

Refrain
Coralie, toi la muse d'install party
Viendras-tu cet été
Au prochain LSM ?
Coralie, ma jolie copinedegeek
As-tu remarqué que « LSM »
Ça rime avec « je t'aime » ?

Après t'avoir chargé plein de patches
Et fini de recompiler le tout (ouuuh-ouuuh)
J'ai bien vu dans le regard des autres
Qu'ils étaient tous jaloux
Parce que mon nouveau dual core
Compilait bien plus vite
(ouuuh-... ah non, ça rime pas !)
Ils disaient à qui voulait l'entendre
Qu'ça compensait ma p'tite...
Bande passante

Refrain
Coralie, toi la muse d'install party
Viendras-tu cet été
Au prochain LSM ?
Coralie, peux-tu mieux régler ton filtre ?
Tous les mails que je t'envoie,
Ce ne sont pas des spams

Après une soirée de folie,
À remplir des bug reports
On a joué sur les routeurs
À recâbler tous les ports
Et quand on a éteint les lumières
Quand tout l'monde est enfin parti
Dans le noir, on a vu clignoter toutes les LED,
C'était tellement joli...

Refrain
Coralie, ma geekett'goth endurcie
Es-tu sûre de m'avoir donné
La bonne adresse e-mail ?
Coralie, j'veux pas croire qu'c'est du mépris
Mais je suis encore bloqué
Par ton client IM

Sur le chemin du retour,
On trollait comme des fous (ouuuh-ouuuh)
Tu me disais qu'une Slackware
C'était mieux qu'Ubuntu
Mais ça me posait un gros problème
Pour te passer mes programmes (ouuuh-am)
Car on sait bien qu'une Slackware
Ne lit pas les paquets Debian

Refrain
Coralie, je n'suis pas un abruti
J'ai bien vu que l'adresse
Que tu m'as donnée est morte,
Coralie, ne m'dis pas qu'c'est ton mari
Qui se venge sur moi
En floodant tous mes ports

Refrain
Coralie, j'crois qu'cette histoir'est finie
Est-ce toi qui as posté
Tous mes mails sur bashfr ?
Coralie, arrête ça je t'en supplie
Tous les copains du chan
Se moquent de moi sans cesse.

La chanson Nathalie d'Oldelaf et Monsieur D :
http://oldelafetmonsieurd.free.fr/videos.htm *ou*
http://youtube.com/watch?v=1z9BEofqS_4

Denis Bodor

Utilisation avancée de GNU Screen

Vous connaissez sans doute GNU Screen, le multiplexeur de terminal permettant une utilisation souple et dynamique d'une console locale, X11 ou via SSH. Mais avez-vous déjà parcouru la documentation et remarqué le nombre incroyable de fonctionnalités que GNU Screen offre en plus de l'utilisation basique ?

J'ai découvert GNU Screen il y a fort longtemps après avoir posé une simple question sur un canal IRC : « mais comment fais-tu pour rester connecté 24h/24 tout en changeant d'ordinateur ? ». La réponse tenait bien entendu en un simple mot, la commande `screen`.

I. B.A. BA

L'utilisation de `screen` couplé à `irssi` n'est qu'un aspect de la fonctionnalité principale de l'utilitaire : la capacité à gérer un ou plusieurs écrans en mode texte/console. Il est ainsi possible de disposer de plusieurs terminaux virtuels sur une machine n'ayant qu'une seule console (aucun ou un seul `getty`, ou pire, une console série). La page de manuel décrit GNU Screen comme « un gestionnaire de fenêtres plein écran multiplexant un terminal entre plusieurs processus (typiquement, des *shells*) ».

L'utilisation la plus courante de la commande `screen` est donc :

```
% screen
```

Je vous l'accorde, elle était facile. Une fois l'outil lancé et, selon la configuration par défaut, après l'affichage d'un message d'accueil, vous vous retrouvez avec un simple shell. Les commandes transmises à destination de Screen se composent, par défaut, à partir de la combinaison [CTRL]+a suivie d'un caractère ou d'une autre combinaison de touches. Nous avons, par exemple :

- [CTRL]+a c : `create`, créer un nouveau terminal/fenêtre ;
- [CTRL]+a n : `next`, passer à la fenêtre suivante (en bouclant) ;
- [CTRL]+a [CTRL]+a : `alternate`, basculer d'une fenêtre à l'autre ;
- [CTRL]+a w : `windows`, afficher la liste des fenêtres ;
- [CTRL]+a 3 : passer à la fenêtre 3 ;
- [CTRL]+a « : obtenir un menu de sélection de fenêtre.

Ceci nous permet de faire le tour de la gestion de terminaux virtuels. Avec un peu d'entraînement, on arrive facilement à acquérir des réflexes d'utilisation et donc à exploiter au mieux les fonctionnalités. Mais Screen va plus loin, puisque, outre le fait de servir de multiplexeur, il fait office d'une sorte de `nohup` amélioré. Une session Screen peut, en effet, être détachée du terminal (le vrai), puis rattachée :

- `screen` est lancé et nous obtenons un nouveau terminal ;
- nous lançons notre client IRC, notre MTA préféré, etc. ;
- nous détachons le terminal virtuel de la console matériel (ou du Xterm) avec [CTRL]+A d. Screen nous informe du succès de l'opération avec un message : `[detached]` ;
- nous nous déplaçons sur un autre poste ;
- nous obtenons un shell à distance avec OpenSSH ;
- nous rattachons le terminal virtuel à notre nouvelle console avec `screen -r`.

Ainsi, nous pouvons allègrement détacher n'importe quelle session de Screen et la reprendre par la suite. Le fait de détacher le terminal virtuel n'est pas un pré-requis, puisque qu'il est possible de détacher automatiquement une session et la rattacher localement avec les options `-rD`. Chaque session de Screen est identifiable. Vous pouvez donc lancer autant de `screen` qu'il vous semble nécessaire. Cependant, on préfèrera souvent n'en utiliser qu'une en multipliant les fenêtres. Les sessions Screen sont référencées sous la forme `pid.tty.hôte` :

```
% screen -ls
There are screens on:
        27871.pts-6.morgane     (Attached)
        27269.pts-5.morgane     (Detached)
2 Sockets in /var/run/screen/S-denis.
```

Enfin, bien que cela dépasse sensiblement la simple utilisation basique, notons la possibilité de surveiller une fenêtre pour y détecter une éventuelle activité. Ainsi, l'utilisation de la combinaison [CTRL]+A, puis M comme Monitor déclenchera la surveillance. Nous pouvons alors basculer sur une autre fenêtre et travailler. Si une quelconque activité est détectée, un message s'affichera dans la ligne d'état et nous serons avertis. Idéal pour un client IRC, un téléchargement ou un script un peu lent.

On pourra alors choisir une session spécifique à rattacher localement avec `screen -rD 27871.pts-6.morgane`. On peut choisir de ne pas rattacher la session en utilisant simplement `-d` ou `-D`. Il est également possible de ne pas détacher la session avant de la rattacher localement. Ceci permet une utilisation simultanée du terminal virtuel depuis deux consoles. On parle alors de mode multi-affichage (*Multi display mode*) idéal pour l'assistance

des utilisateurs ou la formation/intervention sur un serveur. L'option à utiliser en lieu et place de `-d` est `-x`.

Plusieurs combinaisons de `-r`, `-R`, `-d` et `-D` sont possibles, mais la plus utile sera sans doute `-DRR`. Elle aura pour effet de faire tout ce qu'il faut pour détacher et rattacher la première session existante sur le système. Il s'agit ici de remplacer les terminaux virtuels `getty` et/ou Xterm par une seule console (ou Xterm) faisant fonctionner Screen.

2. Utilisation plus avancée

À présent, vous connaissez Screen, si ce n'était pas déjà le cas il y a 5 minutes. Nous pouvons donc pousser un peu plus loin. L'idée générale qui se cache derrière cet outil est son utilisation exclusive et systématique. En vous débarrassant de toutes autres formes de terminaux virtuels, vous obtiendrez un environnement pouvant être accessible de partout et pouvant s'adapter très facilement. Un bon exemple se résume en l'utilisation d'un Xterm unique occupant tout l'écran du premier bureau virtuel de votre gestionnaire de fenêtre X (au hasard WindowMaker ou e17). Nous allons voir dans la suite que Screen est amplement suffisant et qu'il vous permettrait même d'éliminer les DockApps, Epplets et autres outils comme Gkrellm. Le tout, accessible depuis une simple liaison OpenSSH (ou Telnet si vous n'avez peur de rien ;).

La personnalisation de votre environnement Screen passe pas la création de votre `~/.screenrc`. Normalement, un fichier de configuration général est déjà présent. Dans le cas d'une distribution Debian Etch, c'est `/etc/screenrc`. En oubliant les lignes de commentaires, la configuration de base est relativement simpliste.

Nous avons tout d'abord la définition d'un paramètre décrivant l'influence des créations et suppressions de fenêtres sur la base `utmp` (la liste des utilisateurs connectés au système est donc visible par `who`).

```
deflogin on
```

On définit un bip visuel (flash de l'écran lors d'un bip système). Un message est affiché dans la ligne d'état de Screen (c'est le message par défaut) :

```
vbell on
vbell_msg "   Wuff  ----  Wuff!!  "
```

Vient ensuite la définition du tampon de défilement. C'est l'équivalent d'un [MAJ]+[PAGE_UP]. Notez cependant que ce tampon n'est pas commun à celui de Xterm par exemple. Il existe un tampon pour chaque fenêtre Screen et celui-ci peut être parcouru en passant en mode copie (voir plus bas) :

```
defscrollback 1024
```

Un certain nombre d'associations de touches sont supprimées et d'autres redéfinies. Les raccourcis spécifiés via `bind` sous-entendent l'utilisation préalable de [CTRL]+a :

```
bind ^k
bind ^\
bind \\ quit
bind K kill
bind I login on
bind O login off
bind } history
```

On désactive ensuite la ligne d'état. Attention, ceci ne signifie pas que les messages ne seront plus affichés, mais simplement qu'ils seront en surimpression et en vidéo inverse du contenu de la fenêtre (en bas). Avec `hardstatus` à `on`, par exemple, une ligne est réservée pour les messages. Il s'agit littéralement d'une émulation de ligne d'état si celle-ci n'existe pas sur le terminal de lancement (capacités `hs`, `ts`, `fs` et/ou `ds` du terminal).

```
hardstatus off
hardstatus string "%h%? users: %u%?"
```

Enfin, nous avons les adaptations à apporter aux capacités des différents profils de terminaux (termcap/terminfo). Ceci vous permet de modifier dynamiquement quelques capacités sans avoir à créer un nouveau profil.

```
termcapinfo vt100 dl=5\E[M
termcapinfo xterm*|rxvt*|kterm*|Eterm* hs:ts=\E]0;:fs=\007:ds=\E]0;\007
termcapinfo xterm*|linux*|rxvt*|Eterm* OP
termcapinfo xterm 'is=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;4;6l'
```

À présent, nous pouvons ajouter nos propres éléments de configuration dans notre ~/.screenrc. La première chose sera de personnaliser la ligne d'état via hardstatus (le tout sur une ligne) :

```
hardstatus alwayslastline "%{+b kw}%H%{kg}|%c|%{ky}%d.%m.%Y|%{kg}%=Lef:
%42` Lmml:%43` db:%44` ebay:%45` df:%46`%-0=%{kw}"
```

Impressionnant, non ? Détaillons un peu et ajoutons ce qui manque en début de ~/.screenrc :

```
startup_message         off
vbell                   off
backtick 42 60 0 ~/bin/cptmailgal.pl ~/Mail/lefinnois.mbox
backtick 43 60 0 ~/bin/cptmailgal.pl ~/Mail/lmml.mbox
backtick 44 60 0 ~/bin/cptmailgal.pl ~/Mail/dbodor.mbox
backtick 45 330 0 ~/bin/cptmailgal.pl ~/Mail/ebay.mbox
backtick 46 330 0 ~/bin/cptmailgal.pl ~/Mail/debianassos.mbox
```

Les deux premières lignes parlent d'elles-mêmes, inutile de s'attarder. La partie intéressante concerne backtick. Cette directive permet de définir des commandes dont le résultat sera substitué à chaque occurrence de %nn` où nn est le numéro spécifié immédiatement après backtick.

Le second paramètre est le nombre de secondes durant lesquelles la sortie de la commande est considérée comme valide. Passé ce délai, la commande est exécutée à nouveau. Le troisième argument est le nombre de secondes pour le rafraîchissement (qui est souvent le double du second paramètre ou 0 comme ici). Enfin, nous avons la commande et ses arguments. Ici, le script cptmailgal.pl permet de décompter les messages présents dans une mbox sans toucher aux horodatages (atime, ctime, mtime) pour ne pas perturber le fonctionnement de Mutt.

Il nous suffit ensuite de composer tranquillement notre hardstatus :

- %{+b kw}%H le nom d'hôte en gras (+b) blanc (w) sur noir (k) ;
- |%c| l'heure en vert (g) sur noir (k) et toujours en gras ;
- %{ky}%d.%m.%Y| la date en jaune (y) sur fond noir (k) au format jour (d), mois numérique (m) et année sur 4 positions (Y) ;
- %{kg}%=Lef:%42` en vert sur fond noir et avec un alignement à droite (%=), notre première variable de substitution (%42`) ;
- Lmml:%43` la seconde ;
- db:%44` la troisième ;
- ebay:%45` la quatrième ;
- df:%46` et la cinquième ;
- %-0=%{kw} puis nous désactivons les attributs (gras, couleurs, etc.) pour terminer la ligne.

Ainsi, nous nous retrouvons avec une ligne d'état colorée et indiquant des informations très utiles. Il est possible de cette manière d'afficher toutes sortes d'informations comme un flux RSS ou des alertes diverses. Les possibilités ne sont limitées que par votre imagination et l'astuce dont vous ferez preuve en écrivant vos scripts.

Dans le même genre que hardstatus, nous avons caption qui permet de contrôler l'intitulé des fenêtres. Dans mon fichier de configuration, j'ai défini ceci :

```
caption always "%{+u wk}%?%-w%?%{rk}/%n %t\%{wk}%?%+w%?"
```

Le premier argument de caption, always, permet de forcer l'affichage de l'intitulé même s'il n'y a qu'une seule fenêtre. Nous avons ensuite quelques séquences d'échappement intéressantes :

%{+u wk}%?%-Lw%? est particulièrement instructif, puisque l'utilisation de %? permet de conditionner l'affichage. Ce qui se trouve entre la première occurrence et la suivante ne s'affichera que si une séquence d'échappement présente « résout » vers une chaîne non vide.

Ici %-w permet d'afficher les numéros et noms de fenêtres jusqu'à la fenêtre courante. Nous affichons donc, si besoin, les informations de toutes les fenêtres à gauche de celle en cours.

%{rk}/%n %t\ concerne la fenêtre courante. Nous utilisons du rouge sur fond noir pour afficher le numéro et l'intitulé de la fenêtre séparés par une espace. Nous ajoutons slash et antislash pour des raisons purement cosmétiques.

%{wk}%?%+w%? permet, enfin, d'afficher les informations sur les autres fenêtres. Nous conditionnons l'affichage avec %? après être repassé en blanc sur fond noir. Nous affichons donc, si nécessaire, les numéros et titres des fenêtres se trouvant à droite de la position courante.

Au final, avec notre hardstatus et notre caption, nous obtenons quelque chose comme ceci :

Concluons en parlant rapidement des possibilités de création de raccourcis personnalisés. Voici un exemple donné dans le fichier de configuration permettant d'utiliser le programme urlview pour capturer les

URL et les ouvrir à la demande de l'utilisateur dans le navigateur par défaut (sur une ligne) :

```
bind ^B eval "hardcopy_append off"
          "hardcopy -h $HOME/.screen-urlview"
          "screen urlview $HOME/.screen-urlview"
```

Nous associons la séquence [CTRL]+A [CTRL]+B avec l'enregistrement du *buffer* courant dans le fichier `~/.screen-urlview`, puis nous lançons `urlview` avec le fichier en argument. Plusieurs points sont remarquables. La directive `hardcopy_append off` permet d'écraser le contenu du précédent fichier pour ne pas se retrouver avec une liste incroyablement longue d'URL.

La commande utilisée pour afficher le menu d'`urlview` est `screen`. En effet, un appel à `screen` depuis `screen` provoque la création d'une nouvelle fenêtre comme [CTRL]+A, puis C. C'est un réflexe à prendre que de procéder ainsi plutôt que de créer une fenêtre pour ensuite lancer le programme. Ceci permet, entre autres choses, d'avoir un titre de fenêtre (dans le `caption`) qui correspond au programme et non simplement « bash ».

Arrêtons là pour l'instant. Ceci ne représente qu'une faible partie de ce qu'il est possible de faire avec `screen`. Si vous souhaitez creuser davantage, consultez bien sûr la page de manuel du programme et recherchez « screenrc » sur le Web. Vous trouverez alors bon nombre de fichiers de configuration dont il est possible de s'inspirer.

Denis Bodor,

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

Brèves de Perl

Sébastien Aperghis-Tramoni

Nordic Perl Workshop 2007

Les 28 et 29 avril aura lieu la cinquième édition du *Nordic Perl Workshop*, la conférence Perl annuelle des pays nordiques, à Copenhague, Danemark. Celle-ci étant multinationale, les présentations sont généralement en grande partie, voire en totalité, en langue anglaise, ouvrant ainsi cette conférence à nombre de personnes provenant d'Europe, d'Amérique du Nord, voire de plus loin encore. Ainsi, se déplaceront des USA Allison Randal, Chromatic et Andy Lester ainsi que d'autres personnalités du monde Perl.

Le tarif d'enregistrement a été fixé à 500 couronnes danoises. Le tarif en euros n'a pas encore été fixé, mais devrait se situer entre 60 et 70 euros.

Lien : ▶ http://conferences.yapceurope.org/npw2007/

YAPC::Asia et OSDC.tw 2007

Plus loin de nous se déroulera, les 4 et 5 avril, une nouvelle édition de *YAPC::Asia* à Tokyo, Japon. Avec un tarif attractif de 4 000 ¥ (environ 25 euros), cette conférence avec des présentations en anglais et en japonais avait réuni l'an passé plus de 300 personnes. Cette année ne devrait pas être en reste avec une liste de conférenciers comportant de nombreux noms connus : Tatsuhiko Miyagawa (organisateur de l'événement), Audrey Tang, Mark Jason Dominus, Jesse Vincent, Brad Fitzpatrick, Chia-liang Kao, Brian Ingerson, Dave Rolsky et bien d'autres.

La semaine suivante, les 7 et 8 avril, verra un nouvel *OSDC.tw* (*Open Source Developer's Conference at Taiwan*), la conférence qui réunit les développeurs de différents langages de programmation dans une grande réunion œcuménique. Le thème de cette année porte sur les concepts « 2.0 » les plus chauds du moment, en particulier *Programming 2.0* ou comment écrire des Logiciels libres de manière plus efficace. Grâce à la proximité tant géographique que temporelle avec *YAPC::Asia*, on y retrouvera une bonne partie des orateurs précédemment cités.

Liens : ▶ http://tokyo2007.yapcasia.org/
▶ http://osdc.tw

Olivier Saraja

Sunflow, moteur de rendu par illumination globale

Nous avons découvert Yafray dans les précédents numéros de GNU/Linux Magazine, un moteur de rendu qui permet tous les raffinements propres à un résultat photoréaliste. Nous allons maintenant découvrir, selon le même format d'articles, Sunflow, un moteur de rendu également Open Source, dont l'objectif est aussi de produire des images de synthèse de qualité photoréaliste.

Figure 01 : Gumbo, l'éléphant modélisé en patches de Bézier est un peu l'objet mascotte de Sunflow.

Écrit en Java, Sunflow est construit autour d'un noyau de *raytracing* très flexible et son architecture orientée objet le rend facile à enrichir. À l'origine, ce projet a été développé par Christopher Kulla dans le but personnel de découvrir les algorithmes d'illumination globale et les nouveaux modèles de *shading* des surfaces. Mais il a considérablement évolué pour devenir un moteur de rendu robuste et de qualité, ainsi qu'en témoigne l'impressionnante liste de ses fonctionnalités. Qui plus est, même si Sunflow est particulièrement populaire au sein de la communauté des utilisateurs de Blender, différents scripts d'export ont été développés pour d'autres grandes applications de création d'images de synthèse, comme 3DSMax, Maya, XSI ou Cheetah 3D. La richesse de ces exporteurs varie en fonction de l'application visée et de la bonne volonté de leurs développeurs. L'exporteur dédié à Blender, toutefois, a très récemment connu de belles avancées, mais nous y reviendrons plus tard.

Quelle différence entre un exporteur et un moteur de rendu intégré ? Si l'on prend le cas de Blender, par exemple, il dispose de son moteur de rendu natif, sobrement appelé « moteur interne ». Mais il intègre également Yafray ; c'est-à-dire que lorsque Yafray est sélectionné comme moteur de rendu, l'interface de Blender change (en particulier les panneaux relatifs à l'éclairage, au rendu ou au shading) pour refléter les particularités de Yafray. Avec un exporteur, en revanche, la liaison entre Blender et le moteur de rendu choisi se fait au travers de la mise en œuvre d'un script d'export spécifique, qui va se charger de convertir la scène Blender (ses maillages, ses éclairages, ses *shaders*, etc.) en un format lisible et interprétable par le moteur de rendu externe. Il n'y a donc pas de connexion directe entre Blender et le moteur externe, et de la qualité du script dépendent les possibilités du moteur externe exploitables directement depuis Blender.

1. Installation de Sunflow et de son exporteur pour Blender

Sunflow est un moteur de rendu écrit en Java, ce qui le rend disponible pour toutes les plateformes logicielles compatibles avec Java. Java a longtemps eu la mauvaise réputation d'être très lent, et faire le choix de ce langage pour un moteur de rendu (traditionnellement gourmand en temps de calcul et donc fondamentalement lent) peut surprendre. Mais rassurez-vous, Java est aujourd'hui presque aussi rapide qu'un autre langage compilé, et utiliser ce langage pour écrire un moteur de rendu n'est plus considéré comme absurde. Les dernières versions de Java, notamment, autorisent des améliorations de vitesse de rendu qui sont parfois non négligeables (de l'ordre de 20%), aussi sera-t-il intéressant d'installer un jdk récent.

Installation de Java 1.6.0 sur Ubuntu 6.06

Si vous avez eu comme moi du mal à installer Java 1.6.0 sur une distribution Ubuntu, vous trouverez l'URL suivante intéressante : http://www.developpez.net/forums/showthread.php?t=251439.

Des liens rapides vers les téléchargements les plus populaires se trouvent sur la page d'accueil de Sunflow, mais c'est normalement dans la page Downloads qu'il faut se rendre pour télécharger le moteur de rendu. À la date de rédaction de cet article, Sunflow est disponible en version 0.07.2 et l'exporteur en version 0.07.0.

1.1 Télécharger et installer Sunflow

Le téléchargement de Sunflow à partir de la page d'accueil ne posera aucun problème particulier. L'installation se révélera être des plus simples, puisqu'il suffira d'ouvrir l'archive `sunflow-bin-v0.07.2.zip` précédemment récupérée et d'en extraire le contenu à l'endroit de votre choix. Vous devriez obtenir un répertoire `/sunflow` qui contient en particulier un fichier `sunflow.jar`.

Le Lancement de Sunflow se fait en ligne de commande à l'aide de la commande suivante :

```
java -server -Xmx1024M -jar sunflow.jar
```

Il vous faudra toutefois prendre garde de spécifier pour l'option `-Xmx` la quantité de mémoire Ram de votre ordinateur (1 Go dans cet exemple).

Les scènes d'exemples de Sunflow

Vous avez la possibilité de télécharger quelques scènes d'exemples pour faire vos essais avec le moteur de rendu Sunflow. Ouvrir et étudiez les fichiers `.sc` fournis dans un éditeur de texte permettra d'en apprendre beaucoup sur l'usage de Sunflow sans tutorat particulier, à condition d'être curieux.

1.2 Télécharger et installer l'exporteur pour Blender

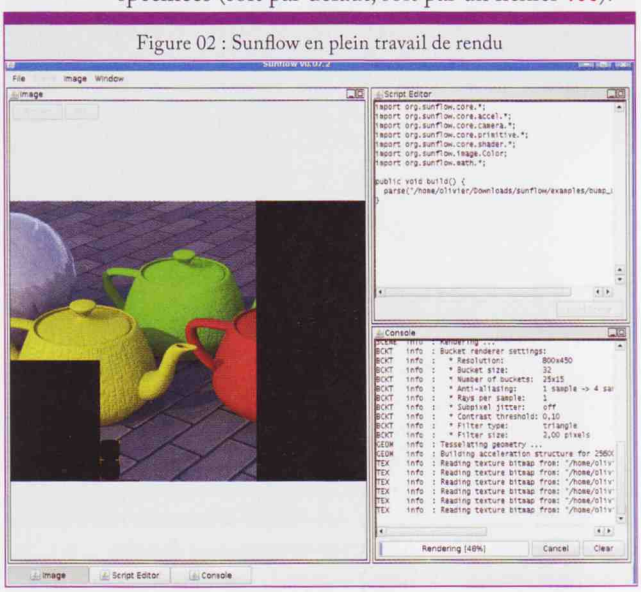
Il est possible de télécharger et d'installer l'exporteur officiel (version correspondant à Sunflow 0.7.0), mais nous ne le recommanderons pas, sauf si vous rencontrez par la suite des difficultés particulières. En effet, il existe un exporteur expérimental (à ce jour intitulé `sunflow_export_merged_13.py`), dont les progrès seront à court terme intégrés dans l'exporteur officiel. Dans la mesure où il permet de spécifier depuis Blender les réglages relatifs à l'illumination globale, nous donnons ici l'URL qui permet son téléchargement:
http://www.geneome.net/blender/blenderfiles/scripts/sunflow_export_merged_13.py

L'installation de l'exporteur est légèrement moins triviale que celle de Sunflow lui-même ; il faut en effet copier le fichier `sunflow_export_merged_13.py` dans le répertoire caché `/.blender/scripts` du répertoire d'installation de Blender. Lors du prochain lancement de Blender, l'option *Sunflow merged_13 (.sc)...* apparaîtra alors dans le menu *File > Export* de Blender. Rien de plus compliqué toutefois !

2. Usage de l'interface de Sunflow

Même s'il s'agit d'un moteur de rendu, Sunflow propose une petite interface permettant de réaliser des opérations simples, comme le rendu lui-même, mais encore de visualiser les options de rendu qui sont spécifiées (soit par défaut, soit par un fichier `.sc`).

Figure 02 : Sunflow en plein travail de rendu



La fenêtre de l'interface est divisée en trois parties :

- *Image* : c'est dans cette partie de la fenêtre que l'image s'affiche progressivement.
- *Script Editor* : cette fenêtre est préchargée avec des éléments par défaut ; vous pouvez alors soit écrire votre scène ici, soit ouvrir un fichier par ailleurs. Vous noterez que le contenu d'un fichier `.sc` n'apparaît pas dans la fenêtre (mail il est fait appel à ce fichier au travers d'une ligne `public void build() {...}`), au contraire du contenu d'un fichier au format `.java`. Si l'intention est louable, elle est limitée par le fait que cet éditeur interne est bien moins confortable qu'un éditeur de texte plus avancé.
- *Console* : c'est ici que Sunflow renseigne l'utilisateur sur toutes les opérations réalisées ; un peu l'équivalent du mode `verbose` de nos chères commandes linuxiennes ;-) À consulter soit par curiosité, soit lorsqu'un rendu ne donne pas du tout les résultats escomptés : de nombreux indices y figurent pour la résolution de tout problème. Fait aussi office de fenêtre de statut, puisque les opérations en cours, les durées de rendu, le taux de progression et bien d'autres choses encore y figurent.

Des boutons en bas de la fenêtre permettent de basculer à tout moment entre les trois parties. Un menu traditionnel occupe la partie supérieure de la fenêtre.

2.1 Les menus

L'interface de rendu de Sunflow présente des menus assez conventionnels, regroupant différentes actions par thèmes voisins.

2.1.1 File

Le contenu de ce menu est très conventionnel : *New* pour créer une nouvelle scène directement à partir du Script Editor ; *Open* pour charger un fichier `.sc` ou `.java` en mémoire ; *Save* pour sauvegarder le fichier `.sc` courant ; *Save As...* pour sauvegarder sous un autre nom le fichier `.sc` courant ; et bien sûr *Exit* pour quitter l'application.

2.1.2 Scene

Build ordonne à Sunflow de reconstruire la scène, équivalant en cela au bouton *Build Scene*. *Build on Open* est une option (active par défaut) qui permet de construire automatiquement la scène lorsqu'un nouveau fichier est chargé.

Render lance le rendu final de l'image, tandis que *IPR* en lance un rendu progressif (sans anti-crénelage) à fins de prévisualisation. *Auto Clear Log* est une option (active par défaut) qui permet de vider la fenêtre Console à chaque nouveau rendu pour en faciliter la consultation.

2.1.3 Image

Reset Zoom permet de remettre l'image à sa dimension originale à l'écran, après usage de la molette de la souris pour la redimensionner, par exemple. *Fit to Window*, pour sa part, réduit ou agrandit l'image de sorte à ce qu'elle soit entièrement visible, quelles que soient les dimensions de la vue Image.

Save Image... permet bien sûr d'enregistrer une image, par défaut au format PNG.

2.1.4 Window

Image, *Script Editor* et *Console* ont la même fonction que les trois boutons du même nom au bas de la fenêtre. Elles permettent d'activer une vue ou l'autre. À noter la fonction *Tile* qui permet de ré-agencer convenablement les trois vues, que ce soit après des opérations manuelles ou une maximisation de la fenêtre.

2.2 La vue Image

Vous pouvez zoomer en avant ou en arrière de l'image grâce à la molette de votre souris ; de même, en maintenant le bouton gauche de la souris appuyé, l'image suit les déplacements de la souris.

2.3 La vue Script Editor

Cette vue présente le script qui permet d'effectuer le rendu d'une image. Les premières lignes renseignent sur les plugins chargés, puis il est soit fait appel à un fichier `.sc` dont le chemin est donné ici, soit la scène est décrite directement avec le langage de description de scène propre à Sunflow. Dans un usage courant à partir d'un exporteur produisant des fichiers `.sc`, cet éditeur est surtout intéressant pour son bouton *Build Scene*. En effet, à chaque fois que vous effectuerez une modification manuelle d'un fichier `.sc`, plutôt que de le charger à nouveau en mémoire au travers du menu *File > Open...*, il vous suffira de cliquer sur ce bouton pour reconstruire la scène dans la mémoire de Sunflow.

2.4 La vue Console

Cette vue indique l'avancée du traitement des données de Sunflow, offrant une vue synthétique des paramètres de rendu qui figurent dans le fichier `.sc`. La consultation de ce log est indispensable à tout rendu sérieux d'une image, car il renseigne sur les anomalies de rendu, l'état en cours du moteur lors d'un rendu, et, par une présentation claire et synthétique des options retenues, permet plus facilement à l'utilisateur d'optimiser sa scène et donc de gagner du temps.

3. L'exporteur pour Blender

Une fois que vous avez composé votre scène sous Blender, vous aurez deux façons de lancer l'exporteur qui génèrera le fichier `.sc` à destination de Sunflow :

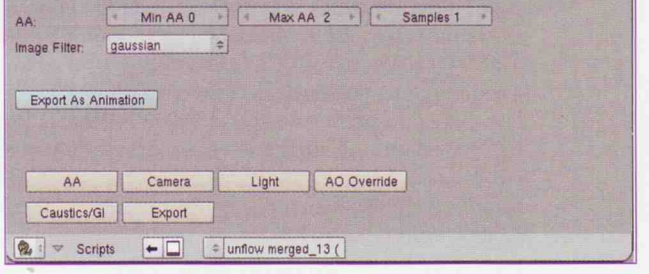
- par le menu de Blender : *File > Export > Sunflow Exporter (.sc)...* (ou *Sunflow merged_13 (.sc)...* suivant ce que vous avez installé) ;
- dans une vue de type Scripts Window : *Scripts > Export > Sunflow Exporter (.sc)...*

Cela aura pour effet d'appeler une interface qui catégorise les options de la scène par familles. Nous aurons l'occasion de revenir sur les différents paramètres ainsi que leur influence sur la scène tout au long de cette mini-série, mais nous insistons sur le fait suivant : l'exporteur ne donne pas encore accès à toutes les possibilités de Sunflow. L'édition manuelle des fichiers `.sc` sera donc indispensable pour un contrôle parfait du contenu de vos scènes. L'exporteur est toutefois suffisamment avancé pour s'amuser sans avoir à ouvrir les fichiers `.sc`, et permettre même quelques raffinements.

3.1 Les options d'anti-crénelage

Ce groupe d'options permet de définir les options relatives à l'anti-crénelage et au type de filtre appliqué à l'image. Attention aux paramètres *Min AA* `0` et *Max AA* `2` proposés par défaut par l'exporteur `Sunflow merged_13.py`. Ils n'ont rien à voir avec les valeurs OSA de Blender, et des valeurs Min `0` et Max `2` sont le plus souvent suffisantes pour vos scènes : les augmenter peut faire exploser les temps de rendu de façon injustifiée.

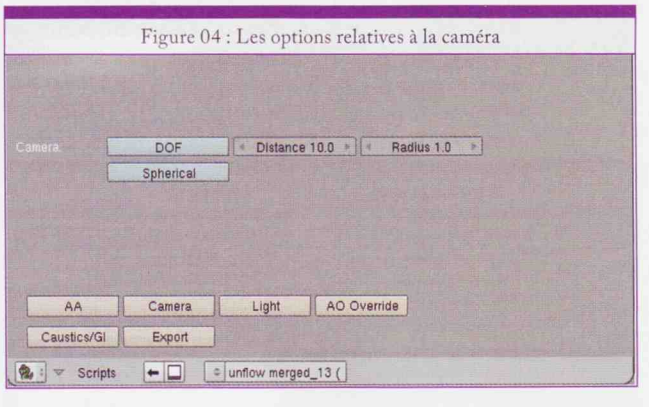
Figure 03 : Les options relatives à l'anti-crénelage



3.2 Les options relatives à la caméra

Les options regroupées ici permettent de spécifier le type de caméra. En activant le bouton *DOF*, vous pouvez mettre en place un effet de flou focal. Le bouton *Spherical*, quant à lui, permet de faire le rendu de la scène en projection sphérique.

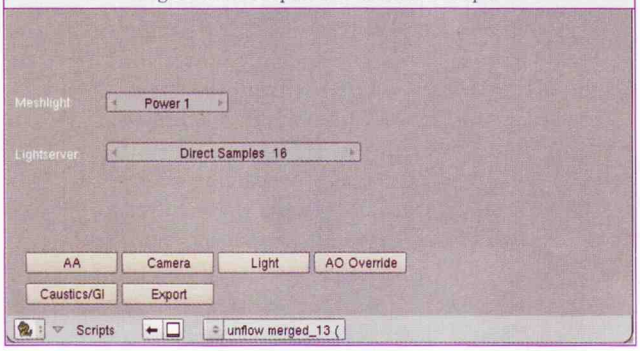
Figure 04 : Les options relatives à la caméra



3.3 Les options relatives aux lampes

À ce jour, seules les lampes de Blender qui sont du type *Lamp*, *Sun* ou *Area* sont supportées. L'option *Meshlight* permet de donner aux lampes un facteur d'intensité : les niveaux d'énergie spécifiés dans Blender étant faibles, il sera souvent nécessaire de commencer en donnant une valeur *Power* de l'ordre de 4 pour commencer tranquillement. L'option *Lightserver* permet de définir le nombre d'échantillons par lampe.

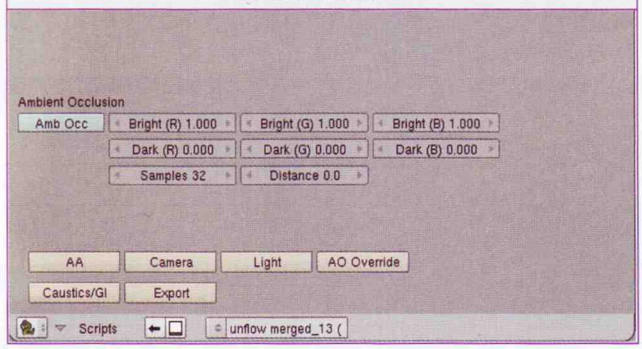
Figure 05 : Les options relatives aux lampes



3.4 Les options relatives à l'occlusion ambiante

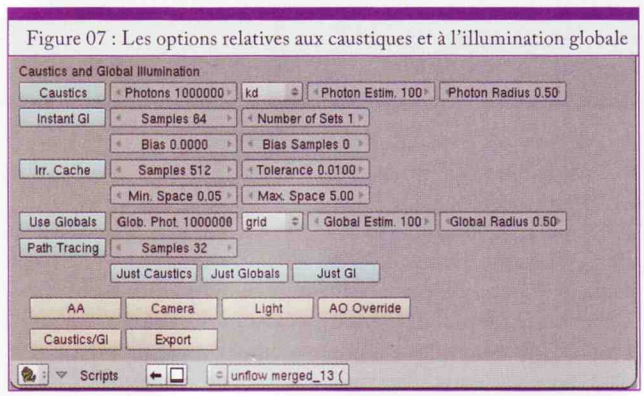
Attention à nouveau, il ne s'agit pas d'occlusion ambiante au sens strict de Blender, mais d'imposer à tous les objets de la scène un shader spécifique qui prendra le pas sur les autres shaders éventuellement déjà mis en place. Ce shader permet de réaliser de rapides prévisualisations de vos scènes tout en offrant un ombrage doux sans pratiquement aucun effort. Les couleurs *Bright* et *Dark* sont tout simplement les couleurs (RGB) claires et sombres d'un pixel, proportionnellement au taux d'occlusion. Les paramètres *Samples* et *Distance* fonctionnent comme dans Blender.

Figure 06 : Les options pour forcer la scène entière à revêtir un shader d'occlusion ambiante



3.5 Les options relatives aux caustiques et à l'illumination globale

Figure 07 : Les options relatives aux caustiques et à l'illumination globale



Ces options sont les plus denses et les plus exhaustives de l'exporteur. La première ligne active la prise en compte des caustiques réfléchis (miroirs) ou réfractés (verre) en définissant le nombre de photons pour la scène. Les autres lignes permettent de spécifier une méthode d'illumination globale au choix parmi *Instant GI*, *Irradiance Caching* et *Path Tracing* (nous reviendrons sur ces trois méthodes dans un prochain article de la mini-série). *Use Globals* est une option de la méthode Irradiance Caching. Vous noterez des options supplémentaires, parmi lesquelles *Just Caustics* qui permet, si le bouton *Caustics* est actif, d'ajouter au rendu les caustiques réfléchis ou réfractés sans pour autant nécessiter de calculer la totalité de la scène en illumination globale.

3.6 L'exportation de la scène

Celle-ci a lieu lorsque le bouton *Export* est pressé : le navigateur de fichiers de Blender fait son apparition, et permet de choisir un chemin, de saisir un nom de fichier se terminant par `.sc`.

Figure 08 : Confirmation de l'export au format .sc

L'export aura pour effet de générer deux fichiers, l'un contenant la géométrie de tous les objets, l'autre contenant les autres informations nécessaires au moteur de rendu pour faire son travail. Les fichiers d'exportation pouvant être très gros (communément plusieurs mégaoctets), il s'avère que la séparation d'une scène en deux fichiers distincts est une solution garantissant la facilité de la manipulation manuelle des fichiers `.sc`.

3.6.1 [fichier].geo.sc

Ce fichier contient les différents objets de votre scène et définit pour chacun d'eux le nom du shader associé, le type de primitives impliquées, le nom de l'objet, le nombre de sommets et les coordonnées de ceux-ci, le nombre de triangles et les combinaisons de sommets permettant de les produire, la définition des normales et les éventuelles coordonnées UV. Pour la scène par défaut de Blender (un cube solitaire au centre de la scène), cela donne le code suivant :

```
object {
          shader "Material.shader"
          type generic-mesh
          name "Cube"
          points 8
               1.0 0.999999940395 -1.0
               1.0 -1.0 -1.0
               -1.00000011921 -0.999999821186 -1.0
               -0.999999642372 1.00000035763 -1.0
               1.00000047684 0.999999463558 1.0
               0.999999344349 -1.00000059605 1.0
               -1.00000035763 -0.999999642372 1.0
```

```
                -0.999999940395 1.0 1.0
        triangles 12
                0 1 2
                0 2 3
                4 7 6
                4 6 5
                0 4 5
                0 5 1
                1 5 6
                1 6 2
                2 6 7
                2 7 3
                4 0 3
                4 3 7
        normals none
        uvs none
}
```

3.6.2 [fichier].sc

Ce fichier contient tous les paramètres de rendu (résolution, niveau d'anti-crénelage, type de filtre), d'illumination globale (méthode d'illumination, nombre de photons, nombre d'échantillons, etc.) ou d'éclairage (type de lampe, puissance de celle-ci, etc.) de la scène. Il présente également tous les shaders (référencés pour chaque objet dans `[fichier].geo.sc`) ainsi que les propriétés de caméra. Pour la scène par défaut de Blender, cela donne cela :

```
image {
    resolution 800 600
    aa 0 2
    filter gaussian
}


shader {
    name def
    type diffuse
    diff  1 1 1
}

shader {
    name "Material.shader"
    type diffuse
    diff { "sRGB nonlinear" 0.800000011921
0.800000011921 0.800000011921 }
}

camera {
    type   pinhole
    eye    7.48113155365 -6.50763988495 5.34366512299
    target 6.82626962662 -5.89697408676 4.89841985703
    up     -0.317370116711 0.312468618155 0.895343244076
    fov    49.1343426412
    aspect 1.33333333333
}

light {
    type point
    color { "sRGB nonlinear" 1.0 1.0 1.0 }
    power 1.0
    p 4.07624530792 1.00545394421 5.90386199951
}

include "test-sunflow.geo.sc"
```

C'est typiquement ce fichier que vous éditerez à la main pour affiner les réglages de la scène, plutôt que de relancer systématiquement l'exporteur de Blender (opération qui peut vite devenir pénible lorsque l'on tâtonne à la recherche des bons paramètres). De plus, certains shaders sont exportés sur la base de quelques paramètres piochés dans les réglages internes à Blender, mais se voient attribuer des paramètres par défaut qu'il n'est possible de changer qu'au travers d'un éditeur de texte. C'est le cas des shaders Phong et Ward, par exemple.

4. Sunflow et Blender par la pratique

La procédure normale de travail avec le couple Blender plus Sunflow consiste à composer sa scène dans Blender, à l'exporter au format `.sc`, à retoucher (si nécessaire) le fichier exporté dans un éditeur de texte, puis à ouvrir le fichier exporté dans l'interface de Sunflow pour en réaliser le rendu.

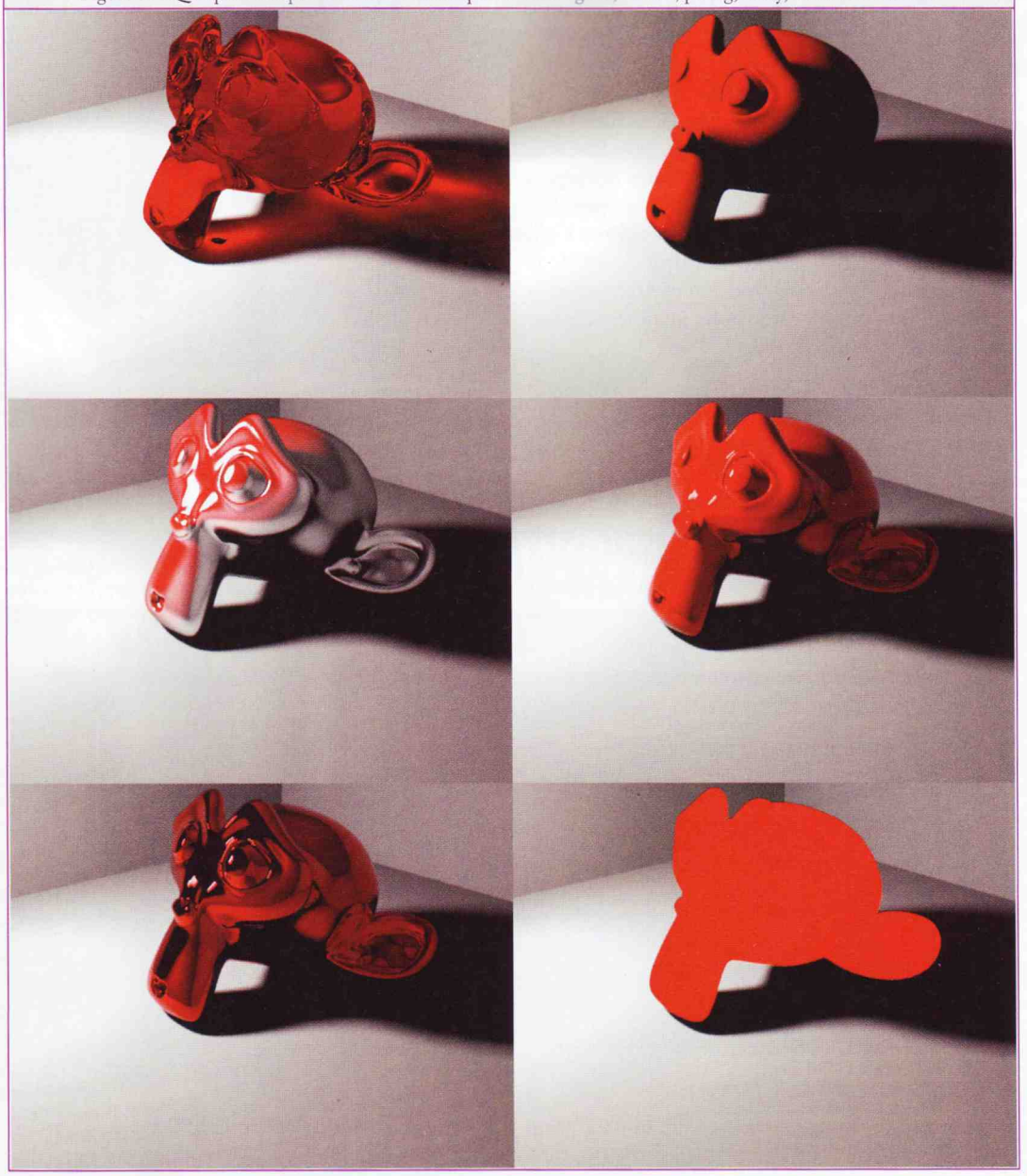
4.1 Sunflow et les matériaux exportés par Blender

La principale subtilité de l'exporteur est que la conversion des matériaux de Blender en shaders équivalents de Sunflow ne se produit pas automatiquement. Certains paramètres de Blender sont effectivement exportés en leur équivalent Sunflow, mais seulement si le nom du matériau commence par une certaine séquence de caractères :

Matériau commence par...	Shader Sunflow correspondant	Paramètres Blender récupérés
`sfamb`	ambient occlusion	bright (Col), dark (Spe)
`sfdif`	diffuse	diff (Col)
`sfpho`	phong	diff (Col), spec (Spe, hard)
`sfwar`	ward	diff (Col), spec (Spe)
`sfshi`	shiny	diff (Col), refl (RayMirr)
`sfmir` et bouton *Ray Mir* actif	mirror	refl (Col)
`sfgla` et bouton *Ray Transp* actif	glass	eta (IOR), color (Col)
`sfcon`	constant	color (Col)

À la lecture de ce tableau, il apparaît évident que seul un faible nombre des paramètres de Blender sont récupérés automatiquement, mais cela est sans doute dû au fait que le système de matériaux de Blender se veut modulaire et générique, tandis que les shaders de Sunflow sont, au contraire, spécialisés. Attention, la plupart de ces shaders sont exportés de façon invisible avec des paramètres supplémentaires par défaut que seule la lecture à l'aide d'un éditeur de texte permettra d'en prendre connaissance et, éventuellement, de les modifier.

Figure 09 : Quelques exemples de shaders fournis par Sunflow : glass, diffuse, phong, shiny, mirror et constant.



Bref lexique des shaders disponibles

Diffuse : il s'agit d'un shader présentant une couleur simple et unique, sans reflets spéculaires.

Ambient Occlusion : ce shader simule l'occlusion ambiante, c'est-à-dire des ombres très douces proches de l'illumination globale, et l'absence de reflets spéculaires.

Glass : ce shader simule le verre.

Mirror : ce shader simule les miroirs parfaits.

Phong : ce shader convient aux matériaux qui ne sont pas des miroirs parfaits, et en fonction des paramètres, aura tendance à produire des reflets plutôt flous.

Ward : encore un shader convenant aux matériaux qui ne sont pas des miroirs parfaits ; en fait, il s'agit d'un shader anisotropique, très semblable à Phong, à ceci près qu'il est possible de conférer une apparence granuleuse à la surface.

Shiny : encore un shader convenant aux matériaux qui ne sont pas des miroirs parfaits ; il permet de doser le niveau de réflectivité du matériau depuis le mat (pas de reflet) jusqu'au miroir parfait.

Constant : ce shader ne tient pas compte de l'ombrage de la surface, et présente la même intensité de luminosité reflétée sur toute sa surface (semblable au mode *Shadeless* de Blender).

4.2 Sunflow et les lampes de Blender

Côté lampes, seules trois des types de lampes de Blender sont actuellement supportés par Sunflow : *Lamp* (type `point`), *Sun* (type `sunsky`)et *Area* (type `area-light`). La notation du niveau d'intensité des lampes est toutefois très différente entre Blender et Sunflow. Par exemple, une lampe de type *Sun* avec une *Energy* égale à `1.000` dans Blender n'éclairera pratiquement rien dans Sunflow. L'un de vos premiers réflexes devra donc être d'aller dans les options *Light* de l'exporteur et d'augmenter la valeur *Power* de la scène pour les lampes de type *Sun* et *Area* ; pour la lampe de type *Lamp*, il vous faudra modifier la valeur à la main dans le fichier `.sc`.

Ce facteur multiplicatif affectera toutes les lampes présentes dans la scène. Il est toutefois difficile de proposer des niveaux de *Power* type, car tout dépend de la taille de la scène, du nombre de lampes, des méthodes d'illumination employées. En particulier, retrouver le même niveau d'illumination que dans Blender peut s'avérer assez difficile, mais c'est là que le mode de rendu *IPR* de l'interface de Sunflow permet de gagner du temps, en affichant assez rapidement une luminosité représentative de celle du rendu final.

Figure 11 : Essai d'habillage de la scène à l'aide de textures images plaquées selon les coordonnées UV exportées depuis Blender

Conclusion

Sunflow est un moteur présentant des modèles d'illumination avancés et très robustes, et ses nombreux autres raffinements valent sans hésitation d'accorder un peu de temps à leur étude. Certes, le système de shaders en est encore à ses débuts, mais dans la mesure où il n'est pas prévu pour être intégré à Blender et que la plupart des applications 3D disposent de leurs propres algorithmes de placage de texture, le recours systématique au dépliage UV devient difficilement contournable, et limite donc l'impact de ce défaut.

Figure 10 : Avec l'énergie par défaut de la lampe unique de la scène, le rendu est totalement noir ; avec Power 50, il reste sombre ; dans ce cas, il a fallu monter à Power 300 pour avoir un éclairage satisfaisant.

4.3 Sunflow et les textures

Le support des textures dans Sunflow n'est pour l'instant pas finalisé. En particulier, il n'y a pas de texture procédurale, et encore moins qui émule celles de Blender. Pour cela, le développeur de Sunflow travaille sur un système de plugin qui permettra à l'utilisateur d'écrire ses propres shaders en Java ; ce sont les shaders janino.

Côté texture image, la meilleure solution consiste à utiliser le dépliage UV, qui est en revanche parfaitement supporté. C'est, à ce jour, la meilleure méthode pour utiliser le placage de texture dans Sunflow. Pour mettre en œuvre cette solution, il est impératif que le modèle ait été déplié (cette opération est désormais très simple dans Blender depuis l'ajout des méthodes de dépliage *Conformal* et *Angle Based* avec la possibilité de spécifier des coutures), qu'une texture de type *Image* ait été attribuée à l'un des canaux de texture, et que ce canal spécifie *UV* comme *Map Input*.

Pour le reste, Sunflow tient le haut du pavé, surtout en illumination globale ; les rendus sont de qualité et relativement rapides, surtout si l'on en revient au fait que l'application est écrite en Java. Son système d'anti-crénelage offre également des résultats très propres, et il est rarement nécessaire de pousser les paramètres au-delà de ceux proposés par défaut par l'exporteur `sunflow_export_merged_13.py` (qui diffèrent de ceux de l'exporteur officiel).

Bref, Sunflow est plein de bonnes promesses, et nous verrons dans les prochains articles comment l'aider à les tenir toutes et produire, à notre niveau, des rendus d'une grande qualité.

Olivier Saraja,

olivier.saraja@linuxgraphic.org

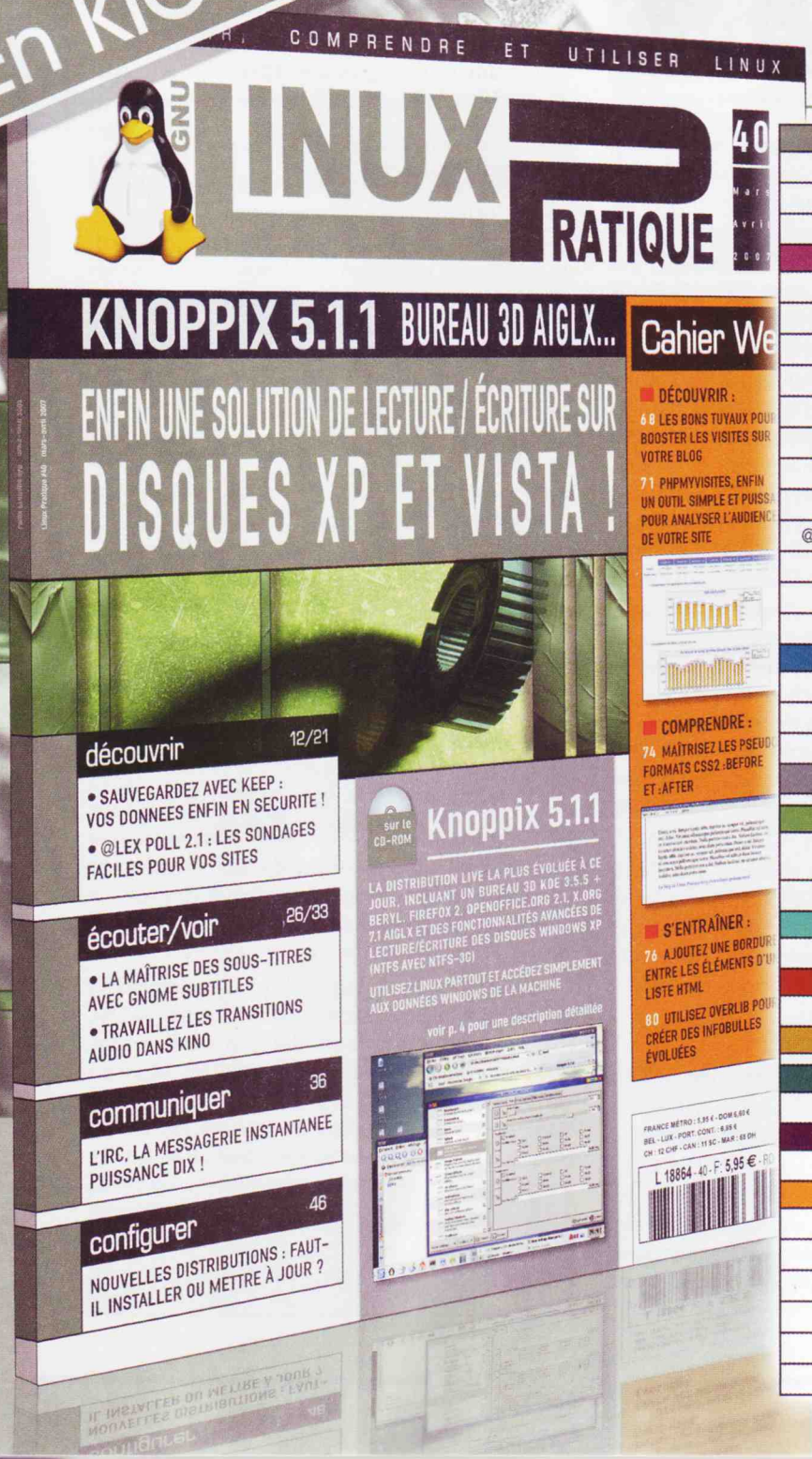
Disponible chez votre
marchand de journaux
et sur
http://www.ed-diamond.com

Apprivoisez votre pingouin !

GNU LINUX PRATIQUE

En KIOSQUE

N°40



Denis Bodor

De la mauvaise gestion des cookies en PHP

Votre site marchand « fait maison » est très convivial. Il sait reconnaître l'internaute qui est déjà venu commander sans que celui-ci n'ait besoin de s'authentifier. C'est fantastique ! Webmaster et développeur PHP avertis, vous maîtrisez totalement la gestion de sessions et les cookies. Mais êtes-vous sûr de ce que vous avez fait ?

Le vol de session est une des techniques les plus faciles à mettre en œuvre pour un attaquant. Le but est simple : se faire passer pour un utilisateur légitime et reconnu sur un site Web utilisant un *cookie* en guise d'identification pour chaque visiteur. Le risque n'est pas ici la compromission du système, ni même le détournement de l'authentification, mais le vol d'informations à propos des visiteurs d'un site.

Quel est, en effet, la valeur marchande pour votre concurrent (ou un *spammeur*) d'une base de données contenant les coordonnées complètes (adresse, numéro de téléphone, email, etc.) des clients d'un de vos sites de commerce ? Autre question : par quel coefficient faut-il multiplier cette valeur si cette base contient également l'historique détaillé des commandes pour chaque compte permettant ainsi d'estimer précisément les centres d'intérêts et les motivations de chaque client ? Difficile de répondre avec exactitude, puisqu'un grand nombre de paramètres entrent en ligne de compte. Cela dépend bien entendu de la taille et de la qualité de la base, ainsi que de la motivation du concurrent ou du spammeur. Cependant, répondre à cette question revient tout simplement à répondre à une autre, pour laquelle vous avez peut-être déjà la réponse : « Combien vaut votre fichier client ? ».

Je vous sens subitement plus attentif. Vous l'avez compris, le risque est donc très important et, comme nous allons le voir dans un instant, l'attaque peut être portée facilement, voire extrêmement facilement si vous avez implémenté votre gestion de session à la légère.

Session et cookies, version simpliste

Le principe de fonctionnement des cookies, s'il est encore nécessaire de le rappeler est le suivant : afin de garder une trace du client, le serveur Web crée une variable qu'il envoie au navigateur. Cette variable est nommée et possède une valeur, une date d'expiration, etc.. L'ensemble forme un cookie. Lorsque le client revient sur le serveur, le cookie est automatiquement présenté.

La seule sécurité réside dans le fait que le navigateur ne divulguera pas un cookie à un serveur qui n'est pas celui spécifié. Bien entendu, c'est un principe de fonctionnement. A l'aide d'un site Web ayant une faille de type XSS, un brin de javascript, il devient possible de subtiliser des cookies. Cela sort du cadre de cet article, puisque ce qui nous intéresse ici est le cas strictement inverse : fournir au serveur un cookie forgé en partant de zéro.

En prenant un langage comme PHP, créer et donner un cookie au navigateur client est un jeu d'enfant :

```
setcookie (Nom, Valeur, Durée_de_vie,
   Répertoire, Domaine, Sécurisé)
```

`Répertoire` peut être spécifié pour restreindre l'utilisation du cookie à un répertoire. Malheureusement, beaucoup de documents informent sur d'éventuels problèmes avec Internet Explorer qui reste encore et toujours largement utilisé (malheureusement !). `Domaine` est le domaine auquel s'applique le cookie. Il est intéressant de remarquer qu'il est parfaitement possible de fournir un cookie au navigateur pour un domaine qui n'est pas le vôtre. Enfin, `Sécurisé`, pouvant valoir `TRUE` ou `FALSE`, permet de spécifier que le cookie ne doit être transmis qu'à travers une connexion sécurisée HTTPS. La version 5.2.0 du langage ajoute une option booléenne `httponly` permettant de faire exactement la même chose que `Sécurisé`, mais pour HTTP (en clair donc).

NOTE

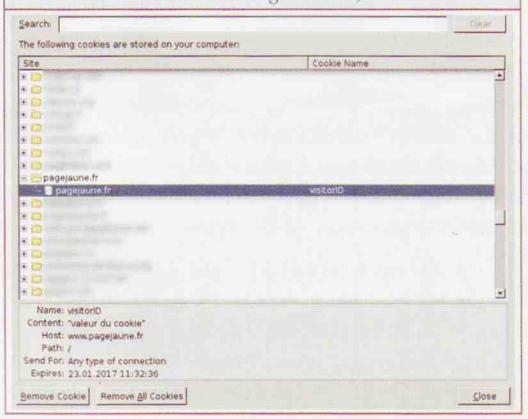
Remarquez que l'utilisation de `setcookie` doit se faire avant toute sortie (avec `print` par exemple). Ceci est une restriction du protocole HTTP et non de PHP, car les informations sur les cookies sont envoyées avec le reste des en-têtes HTTP.

Lire un cookie transmis par le navigateur est également très simple en PHP. Il suffit, en effet, de connaître le nom du cookie pour obtenir sa valeur dans `$HTTP_COOKIE_VARS["nom"];`.

Côté navigateur, l'archivage et l'expiration des cookies sont gérés en interne. Dans le cas de Mozilla FireFox, par exemple, une liste de cookies est maintenue dans `~/.mozilla/firefox/[mon_profil]/cookies.txt`. Il est parfaitement possible d'éditer ce fichier pour changer la valeur des cookies et leur date d'expiration, même si les premières lignes du fichier sont relativemen explicites :

```
# HTTP Cookie File
# http://www.netscape.com/newsref/std/cookie_spec.html
# This is a generated file!  Do not edit.
# To delete cookies, use the Cookie Manager.
[...]
```

Firefox dispose d'un gestionnaire de cookies qui vous permet de prendre connaissance de ce que stocke votre navigateur et éventuellement de supprimer tout ou partie de ces informations. La lecture de la valeur et du nom de chaque cookie est un passe-temps parfois fort amusant. Les chaînes de caractères « valeur du cookie » sont moins rares qu'on ne le pense et ne sont pas sans rappeler certains commentaires de code du genre «int ; // variable i»



Tripatouiller ce fichier, c'est prendre le risque de voir Firefox adopter un comportement erratique (comment ça, ça ne peut pas être pire ?). Bien entendu, des langages comme Perl permettent de manipuler plus simplement ces objets comme nous le verrons plus loin.

L'erreur la plus évidente dans le cadre de l'utilisation de cookies pour la gestion de sessions tient dans le fait de transmettre au navigateur une valeur qui sera directement utilisée pour identifier une personne ou une session. Ainsi, dans les implémentations « maison » de boutiques, de galeries ou autres, il n'est pas rare de voir des choses comme :

```
$res = mysql_query("insert into CLIENT values ('',
'pseudo');");
$numclient = mysql_insert_id();
setcookie (NUMCLIENT, $numclient, time() + 604800);
```

En clair, on insère un nouvel enregistrement dans la table `CLIENT` et on récupère la valeur du champ auto-incrémenté et indexé. Bref, le numéro de client. On utilise ensuite `setcookie` pour transmettre cette valeur au navigateur avec un nom explicite et une date d'expiration définie dans 7 jours (24*60*60*7). Lorsque le client repassera sur le site, il n'aura pas à s'authentifier, puisque le cookie nous est présenté. Nous n'avons qu'à éventuellement vérifier son existence dans la table `CLIENT` et en profiter pour le saluer par son nom.

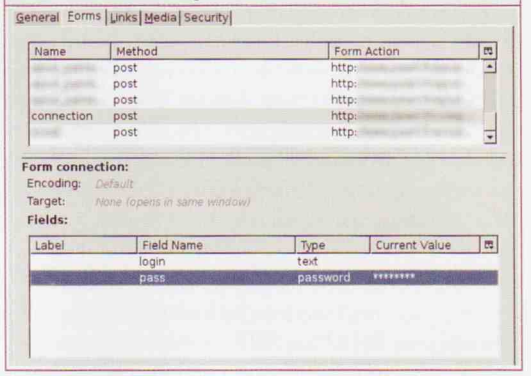
Chose amusante, il suffit que le client édite son fichier de cookies, puis incrémente ou décrémente la valeur de `NUMCLIENT` pour se retrouver identifié comme quelqu'un d'autre. Vous devinez sans peine les conséquences de ce type d'implémentations. Si vous avez le moindre doute, voici de quoi vous faire clairement comprendre le risque :

```
#!/usr/bin/perl
use LWP;
use HTTP::Cookies;
use HTML::Form;
# numéro en argument (valeur du cookie)
$data = shift;
# Nouveau conteneur à cookies
my $coo = HTTP::Cookies->new();
# Ajout d'un cookie dans le conteneur
$coo->set_cookie( "1.0", "NUMCLIENT", $data, "/",
        "monsite.fr", "80", 0, 0, 3600, 0, {} );
# URL de la page avec les informations utiles
my $url = 'http://monsite.fr/compte.php';
# Exécution
my $response = $browser->get( $url );
die "Can't get $url -- ", $response->status_line
  unless $response->is_success;
# est-ce du  HTML ?
die "Need HTML, not ", $response->content_type
  unless $response->content_type eq 'text/html';
# Récupération des formulaires
my @forms = HTML::Form->parse( $response->content, $url );
# Récupération des champs du premier formulaire
my $nom = $forms[0]->find_input('nom')->value;
my $email = $forms[0]->find_input('email')->value;
# Affichage
print "email : $email - nom : $nom\n";
```

Ce simplissime script Perl prend en argument une valeur à attribuer au cookie `NUMCLIENT`, puis accède à l'URL donnée. Il s'agit de la page récapitulative concernant un compte utilisateur, « Mon profil » ou quelque chose de ce genre. Comme nous avons un cookie valide, le serveur présente alors les informations dans un formulaire HTML. La méthode `HTML::Form->parse` nous permet de faciliter le traitement des formulaires sans avoir à *parser* les données HTML nous-mêmes. En quelques lignes, nous affichons deux des informations concernant le profil associé au cookie.

L'attaquant dispose d'un outil très puissant pour faciliter la construction de ce type de script : le navigateur Firefox (j'ai dit « attaquant », pas « script *kiddie* pré-pubère »). Je plaisante, bien sûr, mais un bon navigateur peut présenter des avantages comme l'affichage ordonné des cookies et des informations complètes sur les formulaires.

Firefox informe clairement l'utilisateur des formulaires présents sur la page, ainsi que les différents champs qui composent chacun d'eux.



Le formulaire présentant les informations disponibles afin que l'utilisateur puisse les mettre à jour n'est pas un problème en soi. C'est la possibilité qu'a un utilisateur

d'accéder à d'autres données que les siennes qui l'est. En revanche, il est un cas extrême où la médiocrité frise la caricature : l'utilisation de `value=` dans un champ de type mot de passe (`type="password"`). Rappelons que ce type de champ n'offre absolument aucune sécurité, il ne fait que demander au navigateur de ne pas afficher le texte tapé et de le remplacer par des astérisques. Firefox se prête au jeu en masquant même la valeur du champ dans la fenêtre d'information sur les formulaires. Cependant, le mot de passe est effectivement présent dans la page et donc dans les données HTML. Un simple affichage des « sources » HTML ([CTRL]+[U]) et vous le constaterez par vous-même.

Laisser traîner un mot de passe dans un formulaire est extrêmement dangereux. Non seulement, vous compromettez la sécurité de votre propre service Web, mais également la sécurité des données de vos clients. Nous savons parfaitement que les utilisateurs ont du mal à retenir les mots de passe. Il est donc fort probable que celui utilisé pour accéder à vos services soit identique ou similaire à celui utilisé pour d'autres services (webmail, banque, forums, enchères en ligne, etc.).

Je ne suis peut-être pas assez clair. Reprenons ce que dit la CNIL à ce sujet : « Tout responsable de traitement informatique de données personnelles doit adopter des mesures de sécurité physiques (sécurité des locaux) et logiques (sécurité des systèmes d'information) adaptées à la nature des données et aux risques présentés par le traitement ». De même, « Le non-respect de l'obligation de sécurité est sanctionné de 5 ans d'emprisonnement et de 300 000 euros d'amende (art. 226-17 du code pénal) ». Maintenant suis-je clair ?

Le hash qui ne sert à rien

Le cas précédent était, je vous l'accorde, relativement caricatural. « Implémenter de la sorte une gestion de session n'arrive sans doute pas dans la réalité », me direz-vous. Malheureusement si, mais implémenter naïvement une fonctionnalité est bien peu de chose comparé au fait de se croire à l'abri parce qu'on a ajouté une mesure de sécurité.

Comment faire en sorte que l'attaquant ne puisse pas définir arbitrairement une valeur valide pour notre cookie ? Réponse : il suffit de stocker une valeur dérivée qu'on ne peut utiliser pour déduire la valeur initiale. En clair, le cookie contiendra une valeur calculée à partir de l'information de base et l'attaquant ne pourra pas retrouver cette dernière à partir d'un cookie qui lui a été donné par notre site. Vous l'aurez compris, il s'agit d'un condensé ou *hash* en anglais (hachage en franglais).

Il existe plusieurs fonctions de hachage dont les plus connues sont SHA-1 et MD5. Le développeur Web paresseux (ou incompétent) utilisera très certainement MD5 dans son code PHP, tout simplement car le nom lui évoque quelque chose. Pourtant, en PHP, obtenir un condensé SHA-1 est tout aussi simple que pour MD5 :

```
$str = 'ma chaine';
$md5 = md5($str);
$sha = sha1($str);
```

MD5 peut encore être utilisé pour générer des condensés, mais en aucun cas en guise de hash cryptographique. Une faiblesse dans la fonction a été exposée clairement durant l'été 2004 par des chercheurs chinois. Ceux-ci ont démontré qu'il était possible, sans recherche exhaustive, de découvrir une collision complète (deux données conduisant à la génération de deux condensés identiques). Les spécialistes en cryptographie déconseillent donc l'utilisation de MD5 au bénéfice de SHA-1 ou mieux, SHA-256.

Ici le problème ne réside pas dans la robustesse de la fonction de hachage, ni dans le risque de collision, mais dans le choix des données à fournir à la fonction de hachage. Le cas qui nous intéresse est, bien entendu, le mauvais choix : la valeur précédemment utilisée pour le cookie `NUMCLIENT`. C'est le réflexe stupide par excellence. « Puisque la valeur sur cookie peut être manipulée par un attaquant, je la cache en la remplaçant par un condensé. Me voici protégé »... ou pas.

L'attaquant, voyant un cookie contenant quelque chose comme `ddc7ffd49f740527ba19d43220fb8143` se doutera bien qu'il s'agit d'une représentation hexadécimale d'un condensé de 16 octets. Ensuite, deux solutions :

- Soit, il essaiera de déduire la valeur initiale permettant d'obtenir ce condensé. Là, la première chose qu'il tentera sera sans doute de parcourir les formulaires de la page de profil de l'utilisateur qu'il aura créé à des fins de test. Je pense particulièrement aux champs de type `hidden` pouvant contenir l'information correspondant au condensé. Le test sera ensuite facile pour l'attaquant, puisqu'il lui suffit d'essayer différentes fonctions de hachage sur les valeurs qui lui semblent prometteuses (comme l'identifiant du client sous forme numérique par exemple).
- Soit, il utilisera une attaque en force brute. En d'autres termes, il tentera de générer un condensé correspondant à la valeur du cookie en passant au crible toutes les valeurs possibles. Il pourra également se tourner vers des bases de données valeur/condensé ou des *rainbow tables* (voir http://www.authsecu.com/decrypter-dechiffrer-cracker-hash-md5/).

Cette seconde solution n'est pas aussi gourmande qu'il n'y paraît. Bien entendu, essayer toutes les valeurs possibles pour comparer les condensés est difficilement réalisable... sauf si l'attaquant sait par où commencer. En l'occurrence, s'il s'agit d'une valeur issue de l'insertion d'un enregistrement dans une base de donnée, l'attaque est logique :

```
#!/usr/bin/php
<?
if ($_SERVER['argc'] != 2 ) {
  print "give me hash !\n";
```

```
  exit(0);
  }
$dacook = $_SERVER['argv'][1];
print "Searching for $dacook\n";
for ($i = 1; $i <= 1000000; $i++) {
  if (md5($i) == $dacook) {
    echo "Match ! md5($i) = $dacook\n";
    exit(0);
  }
}
?>
```

Le code ici est en PHP, car l'attaquant utilisera sans doute le même langage que celui utilisé sur le serveur. De plus, si la fonction de hachage est SHA-1, il suffit de remplacer `md5($i)` en `sha($i)`. L'exécution du script avec un condensé en argument est relativement démonstrative :

```
% time ./brutforcecook.php ddc7ffd49f740527ba19d43220fb8143
Searching for ddc7ffd49f740527ba19d43220fb8143
Match ! md5(316785) = ddc7ffd49f740527ba19d43220fb8143
real    0m2.844s
user    0m2.710s
sys     0m0.017s
```

Moins de 3 secondes pour 316 784 condensés calculés avant de trouver la bonne valeur. Cela se passe de tout commentaire. Conclusion, baser une gestion de cession sur un condensé MD5 ou SHA-1 d'une valeur purement numérique ou déductible facilement, c'est mal.

Une fois l'attaquant en mesure de produire des condensés et donc des valeurs valides pour les cookies, il peut renouveler l'opération comme précédemment. En ajoutant cette fois, simplement, la génération du condensé :

```
#!/usr/bin/perl
use LWP;
use HTTP::Cookies;
use HTML::Form;
# numéro en argument (valeur du cookie)
$data = shift;
$digest = md5_hex($data);
# Nouveau conteneur à cookies
my $coo = HTTP::Cookies->new();
# Ajout d'un cookie dans le conteneur
$coo->set_cookie( "1.0", "NUMCLIENT", $digest, "/",
        "monsite.fr", "80", 0, 0, 3600, 0, {} );
# URL de la page avec les informations utiles
my $url = 'http://monsite.fr/compte.php';
# le reste du code est identique
[...]
```

Comme vous le voyez, la modification est ridicule. Se croyant protégé par un condensé impénétrable, l'administrateur du site ou le développeur fera preuve de moins de vigilance alors que la modification n'aurait consommé que quelques minutes au visiteur mal intentionné.

Réduire le risque au niveau du serveur Apache

Pour régler ce genre de problème, la solution est simple : il suffit d'implémenter une gestion de session correcte et, par la même occasion, auditer tout le code du site pour tenter de pallier d'autres problèmes comme les failles XSS (accessoirement, on peut soumettre les développeurs en charge du site au supplice de la roue en guise de formation). Malheureusement, selon la structure et la hiérarchie en place, il n'est peut-être pas possible d'éveiller de manière suffisante un intérêt pour un développement plus intelligent et plus responsable.

Si l'administrateur système se trouve dans une telle (triste) situation, il dispose toutefois de quelques options. En effet, l'attaquant aura sans doute pour objectif de récupérer tout ou partie des informations sur vos clients et leurs achats. En conséquence, il créera un script similaire à ceux présentés ici et utilisera une boucle plus ou moins évoluée pour « pomper » les informations. Il est donc raisonnable de penser qu'il accèdera de manière récurrente à une ou plusieurs pages du site. La différence entre ce « pompage » d'informations et une attaque de type DoS (pour *Denial of Service* ou, en bon français, attaque par déni de service) n'est qu'une question d'échelle temporelle.

Il existe un sympathique module Apache appelé « *Apache Evasive Maneuvers Module* » ou `mod_evasive` permettant de lutter plus ou moins efficacement contre le type d'attaque visant à surcharger un serveur jusqu'à en provoquer l'arrêt. Le principe de fonctionnement est fort simple : le module maintient une table de hachage des adresses IP sources et des URI. Si une combinaison IP/URI (le hash) est présente dans la table, la connexion est rejetée. Un couple IP/URI peut être inséré dans la table sous plusieurs conditions :

- La source demande une URI plus d'un certain nombre de fois par tranche de temps.
- La source nous envoie des demandes plus d'un certain nombre de fois par tranche de temps ;
- La source envoie plus de 50 requêtes simultanées au même processus Apache.

Notez la subtile différence entre les deux premiers points. La tranche de temps est arbitrairement configurable, tout comme le nombre de requêtes par URI et par hôte.

Une fois le module installé sous la forme d'un paquet pour votre distribution où via une installation manuelle avec `apxs2`, celui-ci se configurera très simplement dans votre `/etc/apache2/sites-available/default` (ou tout autre élément/fichier de configuration pour vos sites).

```
<IfModule mod_evasive20.c>
        # Taille de la table.
        DOSHashTableSize 3097
        DOSPageCount 2
        DOSSiteCount 50
        DOSPageInterval 1
        DOSSiteInterval 1
        DOSBlockingPeriod 10
</IfModule>
```

La configuration est basée sur deux types de valeurs. Le `DOSPageCount` pour le décompte de requêtes sur le même URI et le `DOSSiteCount` pour le décompte des

requêtes sur le site complet. Les deux « compteurs » sont associés respectivement à `DOSPageInterval` et `DOSSiteInterval` qui donnent l'intervalle de temps en secondes durant lequel les valeurs limites des compteurs ne doivent pas être dépassées.

Ainsi, cette configuration bloquera :

- l'accès pour une IP à une page si la source dépasse le taux de 2 requêtes par seconde ;
- l'accès à tout le site pour une IP donnée si la source dépasse le taux de 50 requêtes par seconde.

La période de « punition » de l'IP source est définie par la valeur donnée à `DOSBlockingPeriod` (ici 10 secondes). Si, durant cette période, la source fait une autre requête, le compteur de temps est remis à zéro et la punition repart pour la durée spécifiée (10 secondes). L'erreur obtenue dans le cas d'un blocage est `403/Forbidden`.

Il s'agit ici d'une configuration destinée à limiter des attaques DoS. Pour adapter la configuration du module, il faut utiliser des valeurs moins tolérantes, par exemple, 2 requêtes pour 3 secondes. Là, on peut raisonnablement admettre qu'un utilisateur normal ne devrait pas consulter sa page d'informations personnelles avec une fréquence plus importante. La période d'interdiction sera également adaptée et rendue proportionnelle. Une valeur entre quelques 10 minutes et une heure devrait faire l'affaire.

Seul problème, cette configuration a une portée générale pour tout un site. Il n'est pas possible d'utiliser des directives comme `DOSPageCount` dans un contexte `<Location ...>` ou `<Directory ...>`. Il est donc fort peu probable que cela soit réellement compatible avec le site marchand que vous chercherez à protéger. Il peut paraître miraculeux de pouvoir pallier l'incompétence de pseudo-développeurs PHP via la configuration du serveur HTTP, mais cela ne tient pas à grand-chose.

En effet, il suffit de modifier le code de `mod_evasive` de manière à ce qu'il puisse utiliser une expression rationnelle spécifiée de la même manière que les autres paramètres de configuration. Ensuite, avant chaque opération sur la table de hachage, on applique l'expression sur l'URI. Si ça « matche », on applique le reste du processus anti-DOS, sinon, on se comporte comme avec une IP sur liste blanche (directive `DOSWhitelist` de la configuration). Cela se résume en tout et pour tout à une douzaine de lignes de C à ajouter dans les sources du module, en version *quick'n'dirty*. Un développement plus sérieux devra prendre en compte la charge système induite par l'utilisation de `regcomp` et `regexec`.

Remarquez toutefois que si, en tant qu'administrateur compétent, vous avez la possibilité d'adapter le module à vos besoins, vous pouvez tout aussi bien corriger le problème de sécurité lié à la gestion des sessions.

À Tor ou à raison

La détection de ce type d'attaques passe invariablement par le suivi et l'archivage des requêtes. Cela peut aller d'un simple `grep` agrémenté de quelques `cut`, `sort` et `uniq` sur vos fichiers `access.log` à l'IDS en passant par le détournement de modules Apache comme `mod_evasive`. Toute activité suspecte au niveau des pages de profils d'utilisateur peut déclencher une alerte, en particulier en provenance d'un seul et même client.

Cependant, un attaquant potentiel aura vite fait de protéger son anonymat. L'utilisation d'un serveur mandataire (*proxy*) ouvert ou d'une chaîne de serveurs mandataires lui permettra de facilement dissimuler la provenance des requêtes HTTP. Pour peu qu'il utilise correctement le Web, un client IRC ou un lecteur NNTP, il lui sera même possible d'utiliser des réseaux plus importants de serveurs piratés (ou laxistes) ou de machines Windows infectées par des vers ou *bots* (des zombis).

Les infrastructures plus avancées utilisent des ordonnanceurs, permettant de répartir les requêtes entre plusieurs serveurs. Bien entendu, on touche là à des technologies nécessitant un peu plus d'organisation et d'infrastructure que ce dont dispose le délinquant informatique du lycée du coin. On parle ici de crime organisé, de spammeurs de grande taille et de développement de code viral à grande échelle. À vous d'estimer si les informations détaillées concernant vos clients peuvent ou non intéresser ce type d'individus. Pour un forum ou une galerie photo « bricolée », cela serait très étonnant. Pour un site marchand honorant quotidiennement plusieurs dizaines de commandes, on est en droit de se poser la question. Lorsqu'on parle de centaines ou de milliers de commandes ou plus, la question ne se pose plus, on doit se considérer de base, comme une cible potentielle. Bien entendu, à cette échelle, le développement PHP et l'administration des serveurs ne relève *normalement* plus de l'amateurisme.

L'attaquant « bas de gamme » utilisera un proxy ouvert et toutes les connexions sembleront provenir d'une seule source. Le seul avantage qu'il en tirera se résume au fait de dissimuler son IP réelle à notre serveur, mais l'attaque peut être bloquée. De plus, si le serveur mandataire se trouve en France ou dans l'UE, il est sans doute possible de mettre en route une procédure permettant d'obtenir l'adresse IP source par voie légale.

Malheureusement, anonymisation rime souvent avec vie privée. Même si les différentes conférences et documents de l'EFF tentent de démontrer que la défense de la vie privée n'aide en rien à la simplification des activités illégales, le débutant pirate trouvera dans certains outils de quoi l'aider. Je pense, bien entendu, au réseau Tor.

Tor est un réseau d'ordinateurs formant une sorte de VPN ou plus exactement un réseau de tunnels virtuels. Ce réseau est composé de routeurs qui partagent leur bande passante et chiffrent les communications. Résultat, Tor est un réseau où chaque nœud fait transiter une information qu'il ne peut lire, ce qui rend l'analyse de trafic impossible. Utiliser le réseau Tor, c'est utiliser un

point d'accès et encapsuler vos données avant de les faire transiter jusqu'à la destination. C'est juste avant d'atteindre cette destination que vos données quittent le réseau Tor par un point quelconque et imprévisible.

La connexion sur réseau Tor se fait très simplement en utilisant un client pour votre système. Il faut ensuite configurer le client afin de « tor-ifier » les protocoles de votre choix. Manque de chance, HTTP est l'un des protocoles le plus facilement « tor-ifiable ». Avec un système Debian, il suffira ainsi d'installer le paquet `tor` et le serveur mandataire `privoxy`. Une simple ligne dans le fichier `/etc/privoxy/config` et l'affaire est dans le sac :

```
forward-socks4a              /     127.0.0.1:9050
```

Idem pour la configuration du serveur mandataire lui-même :

```
listen-address  127.0.0.1:8118
```

Enfin, la configuration de Tor (`/etc/tor/torc`) est tout aussi simple :

```
SocksPort 9050
SocksListenAddress 127.0.0.1
```

Dès lors, le client Tor pourra se connecter au réseau d'anonymisation et `privoxy` servira de relais pour les connexions HTTP. Les essais peuvent être faits très simplement en définissant une variable d'environnement avec :

```
% export HTTP_PROXY=http://127.0.0.1:8118
```

Il suffit de garder un œil sur les logs d'Apache et de lancer un client Web quelconque (comme eLinks). La connexion semblera provenir d'une autre machine tout comme avec n'importe quel serveur mandataire, à la différence qu'au bout d'un certain temps, le chemin dans le réseau Tor changera et la connexion semblera provenir d'une autre source. On remarquera également que le temps de réponse du réseau Tor n'a rien de commun avec une connexion directe.

Il suffira à l'attaquant de légèrement modifier ses scripts Perl en prenant en compte le serveur mandataire :

```
# Torification proxy
$browser->proxy('http', 'http://127.0.0.1:8118/');
```

Voilà qui devrait vous limiter dans l'analyse des logs, puisque les connexions ne proviennent plus d'une source qu'il est possible de clairement identifier. De plus, la provenance des requêtes n'est pas constante, même s'il reste encore possible d'utiliser le module Apache `mod_evasive`. Mais, là encore, l'attaquant malin trouvera une solution économique et discrète en lançant les requêtes avec intervalle aléatoire :

```
for ((i=6516;i>10000;i-=1))
do
  /chemin/script.pl $i;
  sleep `echo $RANDOM/500*2 | bc`;
done
```

Et ce n'est là qu'une solution basique pour lui. En effet, il peut à loisir intercaler aléatoirement des requêtes « normales », monter en fréquence sur une période relativement importante ou encore utiliser plusieurs machines simultanément. Il n'y a quasiment pas de limite aux techniques permettant de dissimuler les requêtes récurrentes sur une page de votre ou de vos sites. D'une manière ou d'une autre, vous serez toujours perdant.

Une meilleure solution : session_start()

Analyse de l'activité d'un site, blocage des requêtes, filtrage... tout ceci ne constitue pas une solution autre que temporaire (et encore). La seule manière de régler le problème est d'implémenter correctement la gestion de sessions ou, plus simplement, d'utiliser des fonctionnalités éprouvées mises à disposition par la plupart des langages et *frameworks*.

Dans le cas de PHP, la solution s'appelle `session_start()`. Cette simple fonction permet d'initialiser une session :

```
<?php
// page1.php
session_start();
$_SESSION['animal']  = 'chat';
echo '<br /><a href="page2.php">page 2</a>';
echo '<br /><a href="page2.php?' . SID . '">page 2</a>';
?>
```

et

```
<?php
// page2.php
session_start();
echo $_SESSION['animal'];  // chat
?>
```

PHP gère de manière interne les éléments de session et la seule information transmise aux navigateurs sous la forme d'un cookie est un hash totalement indépendant des données stockées. De plus, si le navigateur ne supporte pas les cookies, la gestion de sessions peut tout de même être utilisée via la constante `SID`.

Vous trouverez toutes les informations utiles sur la gestion de sessions en PHP dans la documentation officielle : http://fr.php.net/manual/fr/ref.session.php.

Conclusion

Cette article avait pour objectif de démontrer que l'implémentation « maison » d'une chose aussi triviale que la gestion de sessions peut être bien plus risquée qu'il n'y paraît. Nous avons vu que les connaissances nécessaires pour attaquer un système faible ne sont pas importantes et que le risque, lui, est très important. Avec la prolifération et la globalisation du spam, des données aussi pertinentes que celles d'un fichier client deviennent une véritable mine d'or.

Denis Bodor,

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

Christophe Buffenoir

Introduction à Zope

Un site Web peut aussi bien être statique que dynamique. Dans le premier cas, de simples pages HTML suffisent. Mais que faire dans le second cas ? Il existe différentes technologies, chacune ayant des spécificités, mais aussi un modèle conceptuel différent. Le PHP est l'un des langages les plus utilisés dans ce domaine. Mais qui ne connaît pas la plate-forme de Sun Microsystems ? En effet, le langage Java est très employé dans le monde professionnel. Mais, il existe une alternative : Zope est un framework complet à l'image du J2EE, mis à part qu'il utilise massivement le langage Python.

Ce premier article, après quelques points succincts sur l'installation de Zope, vous présentera l'utilisation de la ZPT. En effet, si le but premier n'est pas d'installer un Zope, mais d'apprendre à l'utiliser, il vous faudra cependant l'avoir à disposition sur votre ordinateur. L'installation en elle-même n'est pas bien compliquée, vous pouvez très bien prendre le paquet de votre distribution ou alors le télécharger sur le site Internet http://www.zope.org. Vous y trouverez également un installateur pour Windows. Nous étudierons seulement la version 2. La version 3 présente une certaine rupture. Seulement, beaucoup de produits Zope n'ont pas encore franchi le cap de la conversion. Aujourd'hui, nous pouvons toujours considérer que Zope 3 n'est pas encore suffisamment répandu pour s'en servir en production.

Pour commencer, installez tout simplement Zope. Dans certaines distributions, il vous sera proposé de créer une instance au moment de l'installation. Si tel n'est pas le cas, il faut le faire soi-même. En effet, une instance, c'est un site. L'instance a son fichier de configuration et son existence propre. Il est possible de comparer l'instance avec l'hôte virtuel d'Apache. Cependant, Apache n'est pas un serveur d'applications contrairement à Zope. Pour créer manuellement une instance, il vous faudra utiliser le script `mkzopeinstance`. Ce dernier est disponible dans le répertoire d'installation de Zope. Il vous sera principalement demandé le répertoire de destination, un nom d'utilisateur et un mot de passe. L'utilisateur qui sera créé aura tous les droits sur cette instance. Nous verrons plus tard comment manipuler l'instance, mais également comment mettre en forme un site Web sans quitter son navigateur Web.

Une fois que vous aurez créé votre instance de Zope, vous pouvez modifier le fichier de configuration de votre instance. Ce dernier contient des informations essentielles, que vous découvrirez très rapidement. Il s'agit, par exemple, du numéro de port sur lequel le serveur écoute ou encore de certains paramètres tels que le support du FTP, le nombre de *threads* actifs, etc. Le numéro de port est défini par la variable `ZOPE_PORT` ou par `HTTPPORT` selon la version de Zope. Attention cependant, il est recommandé de lire entièrement le fichier de configuration. En effet, dans certaines versions, une autre variable s'ajoute au numéro du port. Par exemple, vous avez configuré la variable `ZOPE_PORT` à 8000. Mais votre fichier de configuration fait en sorte que le nombre 80 soit ajouté. Zope écoutera alors sur le port 8080. De plus, de nombreux paramètres sont à prendre en compte dans le cas d'une mise en production. Ce fichier de configuration établit la base de la configuration de l'instance. Dans un environnement en production, il est strictement déconseillé d'utiliser les Zope seuls. Certains outils sont à rajouter pour des raisons de performance et de sécurité. Par exemple, l'outil ZEO vous permettra de configurer un *cluster*. Un *frontend* Apache vous permettra en plus d'allier les fonctionnalités de ce serveur Web à Zope tout en créant un cache essentiel pour les performances. Si, en plus, Apache est configuré avec Squid, vous réussirez à allier la puissance de Zope avec un site Web tout à fait utilisable malgré de nombreuses requêtes.

Avec tout cela, nous nous rendons compte que l'installation d'un Zope pour un environnement en production est une tâche extrêmement difficile. Il faut en effet s'adapter en fonction des restrictions en termes de performance et de bande passante. Malheureusement, les difficultés ne s'arrêtent pas là. Zope est en effet un serveur d'applications et, en tant que tel, embarque certains outils essentiels. Il faudra savoir jouer avec la ZODB. Il s'agit là de la base de données intégrée du serveur d'applications. Certes, elle semble moins rapide et moins performante qu'un PostgreSQL ou qu'un Oracle. Cependant, la ZODB a l'énorme avantage de présenter des données sous forme d'objet. Ainsi, il n'est pas nécessaire d'utiliser un mapping. Mais, pour utiliser cette fonctionnalité, il faut absolument respecter un ratio très restrictif. Il faudra que la ZODB soit utilisée à plus de 80 % pour la lecture des informations. Il ne reste donc que 20 % d'écriture. Néanmoins, vous pouvez toujours utiliser une base de données SQL classique. Pour cela, vous devrez utiliser un connecteur, comme vous le feriez en Java.

Vous découvrirez que Zope fonctionne quasiment uniquement avec de l'objet. Chaque élément de votre site Web est un objet. Nous remarquons alors ici que toute la philosophie de Python est présente dans le serveur d'applications Zope. Vous ne concevrez pas un site Web sur Zope comme vous le feriez en procédural avec PHP. Tout d'abord, Zope implémente le principe du MVC. Il s'agit du modèle de développement « Modèle – Vue – Contrôleur ». Afin de visualiser l'ensemble des objets de votre serveur d'applications, il faut vous rendre sur la page **http://votresite/manage**. Vous verrez alors apparaître, après vous être identifié, un arbre contenant divers items. Vous pourrez, ici, créer des pages Web, insérer des produits Zope, mais aussi créer des répertoires et ajouter des fichiers. Chaque objet, quel que soit son type, a pour principale obligation d'avoir un identifiant clair et unique. C'est ici que vous vous rendez compte des avantages de Zope. L'interface reste assez intuitive pour un informaticien et vous permet de gérer l'intégralité des données. Mieux que cela, en parcourant l'arbre, vous pourrez changer les propriétés de chaque objet et ainsi définir certains paramètres de configuration propres à votre site Web.

Première page avec Zope

Nous allons maintenant ajouter un modèle de page. Ce modèle sera écrit dans le langage ZPT. Le langage en lui-même vous rappellera évidemment HTML. Cela est tout à fait normal, puisque la spécificité de ce langage est de rajouter des attributs aux balises HTML. En effet, ZPT signifie *Zope Page Template*. Nous allons donc créer un modèle HTML que nous rendrons dynamique grâce à une structure dans les attributs des balises. Il s'agit en fait d'un langage au-dessus du HTML. Nous pourrons en effet réaliser des boucles, des structures conditionnelles, des définitions de variables, etc. En bref, nous pouvons ici gérer la plupart des cas que nous ferions dans un langage traditionnel.

L'ensemble du code fourni ici peut être testé sur votre serveur Zope. Pour chaque page, vous devrez aller dans l'interface d'administration et créer un fichier de type `Page Template`. Vous trouverez ce type dans la *combobox* située en haut à droite. La première vous permettra de vous déconnecter, mais la seconde permet de créer un nouvel objet dans le répertoire courant. Une fois que vous aurez créé votre page, en donnant son nom dans l'ID, vous pourrez y accéder avec votre navigateur. Si vous voulez l'éditer, retourner dans l'interface d'administration et cliquez sur son nom. Vous accéderez alors à l'éditeur interne de Zope. Il vous suffira alors de copier le texte et de cliquer sur le bouton permettant la sauvegarde de votre travail. Si vous vous êtes trompé, et qu'une erreur grave empêche l'exécution de votre page, un message d'erreur apparaîtra. Il vous faudra alors corriger votre code pour sauvegarder. Cependant, si vous souhaitez utiliser un éditeur de texte autre, il vous est possible soit de charger le fichier dans l'interface, soit de l'envoyer par FTP serveur.

À chaque balise HTML, nous pouvons ajouter un attribut `tal`. C'est ce qu'on appelle une opération TAL. La syntaxe exacte est la suivante :

```
<balise attributs tal:operation="expression">contenu</balise>
```

Évidemment, les termes opérations et expressions sont à remplacer par quelque chose de plus significatif pour Zope. Sept opérations différentes peuvent être effectuées. La première, `content`, remplace le contenu de la balise lors de l'exécution. La seconde, `replace`, permet de remplacer l'ensemble de la balise. La troisième, `attributes`, s'occupe de la manipulation des attributs de la balise. La quatrième, `define`, sert à définir une nouvelle variable. La cinquième, `repeat`, nous servira pour effectuer une boucle. La sixième, `omit-tag`, supprime la balise lors de l'exécution. Cette opération sera très utile pour des commentaires ou désactiver une certaine partie de la page. Enfin, la dernière opération, `on-error`, définit le comportement en cas d'erreur sur la balise.

Nous voyons alors apparaître les différentes structures nécessaires pour manipuler notre page Web. Nous avons des boucles pour, par exemple, afficher le contenu d'un caddie. Les variables nous permettront de manipuler des données. Mais, en plus, nous avons ici des outils pour agir directement sur l'affichage, en modifiant, en remplaçant ou en supprimant des éléments HTML. Cependant, il nous faut également de quoi manipuler et, surtout, accéder aux différentes données. C'est le rôle des expressions TALES. Mieux encore, le langage METAL nous permettra d'utiliser des macros au sein de la page HTML. Effectivement, ce fonctionnement ressemble très étrangement aux pages JSP, tout en y étant incompatible.

Nous allons maintenant écrire notre première page à l'aide de ZPT. Le but en est très simple : nous allons prendre une liste Python de titres et, pour chaque titre, nous créerons un paragraphe identique. Nous mettrons évidemment un mot ou une phrase bidon afin de visualiser le résultat. Nous verrons ainsi comment créer une boucle sur un bloc important de données. Il faudra appliquer exactement le même principe pour afficher le contenu d'un catalogue de magasin en ligne. Une balise mère contiendra l'attribut `tal`, et c'est elle qui nous permettra de boucler. Son contenu sera formé d'autres balises qui, elles-mêmes, pourront contenir des attributs `tal`, et exécuter ainsi des commandes.

```
<html>
  <head>
    <title>Test d'une boucle</title>
  </head>
  <body tal:define="titres python:['Premier titre',
'Deuxième titre', 'Troisième titre', 'Quatrième titre']">
    <h1>Test d'une boucle en ZPT</h1>
    <div tal:repeat="chapitre titres">
      <h2 tal:content="chapitre">Titre du chapitre</h2>
      <p>Le texte <span tal:content="chapitre">Titre
du chapitre</span> a été inséré automatiquement.</p>
    </div>
  </body>
</html>
```

Nous avons alors défini en premier lieu une liste. Pour cela, nous avons vraiment utilisé un type de données Python. Comme il a été dit plus haut, Zope s'appuie largement sur le langage Python. Ainsi, si vous voulez l'utiliser, il faudra vous familiariser avec les notions de ce langage. En Python, une liste peut contenir n'importe quoi, c'est comme le tableau en C, mis à part que la liste n'a pas de typage. Il s'agit d'un objet `list` pouvant contenir n'importe quels autres objets. Ici, nous l'avons rempli avec les titres, c'est-à-dire des chaînes de caractères. Ensuite, nous avons bouclé dessus et cela a créé une page Web tout à fait dynamique, et ce, sans sortir du modèle de la page.

Le premier attribut `tal` utilise l'opération `define`. Ainsi, nous pouvons définir une variable. Cette variable, nommée `titres`, est une liste Python. Elle contient en réalité quatre titres différents. Plus loin, une balise `div` a un attribut `tal:repeat`. Ce dernier crée une variable appelée `chapitre`, et va permettre d'avoir quatre blocs distincts. Il y aura alors un bloc par valeur dans la liste. Par la suite, dans le bloc, la variable `chapitre` est appelée par des attributs `tal:content`. Le contenu des balises concernées sera alors entièrement remplacé par la valeur de la variable `chapitre`.

Nous allons maintenant réutiliser exactement le même principe, mais pour créer un formulaire. Nous allons reprendre la boucle afin de créer une liste de choix. Il s'agira ici d'options nommées de manière totalement arbitraire. Ainsi, une liste contiendra des chaînes de caractères qui seront insérées dans un `select` multiple. L'utilisateur choisira alors les options qui lui conviennent. Il devra ensuite cliquer sur un bouton « Valider ». Il sera alors réorienté vers une autre page. Cette dernière reprendra quasiment à l'identique le code précédent. Cependant, le texte affiché ne contiendra plus des titres, mais les options précédemment choisies par l'utilisateur. L'exemple peut être simple, mais le principe de base restera le même. Vous trouverez ci-dessous le code pour afficher le formulaire suivi du code de la page de résultats.

```
<html>
  <head>
    <title>Formulaire</title>
  </head>
  <body tal:define="options python:['option1', 'option2', 'option3', 'option4',
'option5']">
    <h1>Un formulaire en ZPT</h1>
    <p>Sélectionnez les options dans la liste ci-dessous :</p>
    <form action="resultat.html" method="post">
      <select name="options" multiple="multiple">
       <option tal:repeat="option options" tal:content="option">Libellé de
l'option</option>
      </select>
      <input name="submit" type="submit" value="Valider"/>
    </form>
  </body>
</html>

<html>
  <head>
    <title>Test d'une boucle</title>
  </head>
  <body>
    <h1>Test d'une boucle en ZPT</h1>
   <div tal:condition="exists:request/form/options" tal:repeat="option request/
form/options">
      <h2 tal:content="option">Libellé de l'option</h2>
     <p>Vous avez sélectionné l'option <span tal:content="option">libellé de
l'option</span>.</p>
   </div>
    <div tal:condition="not:exists:request/form/
options">
      <p>Aucune option n'a été choisie !</p>
    </div>
  </body>
</html>
```

La page de formulaire n'introduit pas réellement de nouvelles notions. Nous réalisons une boucle comme précédemment, mais celle-ci est contenue dans l'option du select multiple. Ainsi, la boucle reproduira autant d'options que de chaînes contenues dans la liste. Le code de la page de résultats est cependant un peu plus complexe. Nous voyons apparaître ici deux blocs différents contenant une condition. Un test est réalisé sur l'existence ou l'inexistence d'une variable. Le nom de la méthode appelée pour vérifier l'existence ne se fait pas exactement comme dans du code Python pur. En effet, le nom de la fonction est suivi de deux points, et le passage de l'argument se fait après. De plus, ici nous n'avons pas défini le nom de la variable. En fait, `request` est un objet disponible pour chaque page. Il contient diverses informations intéressantes comme l'URL de la page, des informations son navigateur ou encore le résultat du formulaire. Ce dernier est inclus dans `form`. Pour appeler une variable d'un formulaire, il suffit de la sélectionner par son nom. Le type de cette variable dépend réellement du contrôle utilisé dans le formulaire. Comme nous avons utilisé un `select` multiple, nous obtenons une liste de chaînes. Une variable n'existe que si le contrôle associé a été rempli dans le formulaire. C'est-à-dire que si l'utilisateur n'a

Abonnez-vous !

11 Numéros de Linux Magazine

1 an de bonne lecture, bien UNIX, bien technique... Bref...

Les 3 bonnes raisons de vous abonner !

- Ne manquez plus aucun numéro
- Recevez Linux Magazine chaque mois chez vous, ou dans votre entreprise
- Economisez 15,20 €/an ! (soit plus de 2 magazines offerts !)

→ Des offres de couplage sont disponibles
→ Retrouvez les Tarifs étrangers hors France Métro sur www.ed-diamond.com

BON D'ABONNEMENT À REMPLIR ET À RETOURNER À (OU PHOTOCOPIE)

LINUX MAGAZINE - BP 20142 - 67603 SELESTAT CEDEX

11 Numéros de Linux Magazine à 53 € Offre France Métro

Soit Votre Linux Magazine à 4,82 € (Tarif au numéro dans le cadre d'un abonnement France Métro)

Pour les tarifs étrangers, consultez notre site : www.ed-diamond.com

Les 4 façons de vous abonner !

» Par courrier postal en nous renvoyant le bon ci-dessous.

» Par le Web, sur notre site : www.ed-diamond.com.

» Par téléphone (paiement C.B.) entre 9h-12h & 15h -16h au 03 88 58 02 08.

» Par Fax au 03 88 58 02 09 C.B. et/ou bon de commande administratif

◎ OUI, je souhaite m'abonner à Linux Magazine pour 11 numéros

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

2 Je joins mon réglement :

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

Offres collectionneurs

LES ANCIENS NUMÉROS !

TOUJOURS DISPONIBLES !

LES 4 FAÇONS DE COMMANDER !

- PAR COURRIER POSTAL EN NOUS RENVOYANT LE BON CI-DESSOUS.
- PAR LE WEB, SUR NOTRE SITE : WWW.ED-DIAMOND.COM.
- PAR TÉLÉPHONE (PAIEMENT C.B.) ENTRE 9H-12H & 15H -18H AU 03 88 58 02 08.
- PAR FAX AU 03 88 58 02 09 C.B. ET/OU BON DE COMMANDE ADMINISTRATIF

BON D'ABONNEMENT À REMPLIR ET À RETOURNER À (OU PHOTOCOPIE)

Bon de commande Linux Magazine

RÉFÉRENCE	Prix / N°s	Qté.	Total
Linux Magazine 83 Greylist Eliminez le SPAM à la racine	5,95 €		
Linux Magazine 84 Déploiement de hotspots Wifi sécurisés	5,95 €		
Linux Magazine 85 Firewall: Netfilter & NuFW	6,20 €		
Linux Magazine 86 Serveur SMTP: Routage des mails avec Postfix	6,20 €		
Linux Magazine 87 Le point sur Mono .NET Java et les Brevets	6,20 €		
Linux Magazine 88 Sécurité: Smartcards & Tokens	6,20 €		
Linux Magazine 89 Utilisation avancée de XEN	6,20 €		
Linux Magazine 90 ASTERISK Le serveur de téléphonie IP	6,20 €		
Linux Magazine 91 AJAX AVANCÉ Principe, fonctionnement et pièges	6,20 €		
Linux Magazine 92 Paravirtualisation XEN & SLO Répartition de charge	6,20 €		

Bon de commande Linux Magazine Hors Série

LM HS 12 Firewall votre meilleur ennemi Acte 1	5,95 €		
LM HS 13 Firewall votre meilleur ennemi Acte 2	5,95 €		
LM HS 15 GIMP et la Photo	5,95 €		
LM HS 16 Kernel (1)	5,95 €		
LM HS 17 Kernel (2)	5,95 €		
LM HS 18 Haute Disponibilité	5,95 €		
LM HS 19 The Gimp 2.0	5,95 €		
LM HS 20 PHP 5	5,95 €		
LM HS 21 Recyclez vos PC	6,40 €		
LM HS 22 GIMP et le Web	6,40 €		
LM HS 23 Linux et electronique	6,40 €		
LM HS 24 Linux Embarqué	6,40 €		
LM HS 25 Linux Embarqué 2	6,40 €		
LM HS 26 Spécial The GIMP	6,40 €		
LM HS 27 Électronique et Linux	6,40 €		
LM HS 28 Administration systeme avec Debian	6,40 €		
	sous TOTAL		
	Frais de port France Metro		+ 3,81 €
	Frais de port Etranger		+ 5,34 €
	TOTAL		

LINUX MAGAZINE - BP 20142 - 67603 SELESTAT CEDEX

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

GNU LINUX MAGAZINE / FRANCE

2 Je joins mon réglement :

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

sélectionné aucune option, la variable n'existe pas. Tenter de boucler sur la variable provoquera une erreur, et affichera un message à l'écran. Il faut donc absolument tester l'existence de la variable avant de la manipuler. Ici, un message différent s'affiche si la variable n'existe pas. Par contre, si la variable existe, il suffit de faire une simple boucle comme nous l'avons déjà fait pour afficher l'ensemble des valeurs à l'écran.

Bien sûr, nous avons simplement étudié ici la syntaxe des pages ZPT, et nous ne nous sommes pas à intéresser à la récupération des variables dans une base de données ou encore à l'exécution de code Python pour la page Web. Mais ces possibilités existent. Elles seront très certainement l'objet de prochains articles.

Tout au long de cet article, nous avons pu constater à quel point l'écriture de pages dynamiques avec Zope peut être relativement simple. Mais nous avons également vu, au début de cet article, que le serveur est un peu particulier. Il faut réellement installer Zope sur le serveur lui-même. Malheureusement, il n'existe aujourd'hui aucune offre ouverte au grand public pour héberger un site Web avec Zope. Une entreprise française a ouvert un service Objectis pour l'hébergement des sites Web des particuliers. Ce service est entièrement gratuit, mais il a connu un très fort succès. La société a alors décidé de fermer les inscriptions. Depuis, le service est à nouveau ouvert, mais les candidats à l'inscription font l'objet d'un vote. Ainsi, le but est clairement de limiter le nombre d'utilisateurs. La multitude de demandes ne peut être satisfaite par l'unique offre existante. À quand un Zope chez Free ? Il est quand même vraiment très simple de gérer un Plone, système de gestion de contenu réputé, et tournant sous Zope.

Christophe Buffenoir,

Clarisys Informatique – http://www.clarisys.fr
http://www.buffenoir.org

Les variables implicites de ZPT

Zope dispose de plusieurs variables différentes incluses dans ZPT. Elles permettent de connaître l'environnement de l'utilisateur comme du serveur. La première représente l'utilisateur et se nomme « *user* ».

Il s'agit d'un objet contenant différentes méthodes. L'utilisateur représenté est un utilisateur connecté au site Web et ayant certains droits. L'utilisation de cette variable est essentielle dans le cas d'un site Web où différents utilisateurs manipulent des informations importantes. Par exemple, si vous faites un site marchand, l'utilisateur de base pourra acheter, le vendeur pourra vendre et l'administrateur pourra gérer le site. En aucun cas, un utilisateur de base ne pourra vendre un objet ou administrer le site Web.

Un objet `template` représente la classe `Page Template`. Il peut toujours être utile, dans de rares cas, de manipuler directement cette classe. Si vous souhaitez accéder directement au document, vous pourrez utiliser la variable `here`. Un autre objet nommé `root` représente la racine du serveur Zope. Nous aurions pu alors dans cet article, par le biais de cette variable, définir une liste Python dans un autre fichier et y accéder depuis le modèle de page. La variable `request` représente la requête de l'utilisateur. Un autre encadré vous informera des différentes données disponibles à l'intérieur de cet objet. La variable `container` vous permettra d'accéder au conteneur de l'objet `template`. Il s'agit généralement du répertoire. Un autre objet nommé `modules` contient les modules Python utilisables dans la page. En effet, seul un nombre limité de modules est disponible dans ZPT. Vous pourrez également accéder aux attributs de la balise en cours via la variable `attrs`. Les arguments de type `keyword` sont disponibles dans la variable `options`. Vous remarquerez alors que cette variable a été redéfinie dans l'exemple de cet article. Cela empêche toute utilisation de l'objet implicite à l'intérieur des balises où la redéfinition a lieu.

La variable `default` permet de laisser intacte une balise malgré la commande de changement des attributs, du contenu ou de la balise complète. La valeur par défaut est gardée. Enfin, une dernière variable existe. Il s'agit de `nothing`. Comme son nom l'indique, cette valeur permet tout simplement d'effacer le texte.

Le contenu de l'objet request

L'objet `request` contient différentes informations importantes concernant l'environnement de l'utilisateur. Il dispose de quatre catégories de variables. La première s'appelle `form`. Nous l'avons utilisée dans cet article. Elle contient tout simplement l'ensemble des résultats du formulaire qui a appelé cette page. Les variables sont nommées en fonction du nom du contrôle dans le formulaire.

Si le contrôle n'a pas été renseigné, la variable n'est pas créée. La seconde catégorie, nommée `cookies` contient les cookies. Les variables contenues dans `other` ne rentrent pas dans les trois autres catégories. Il s'agit notamment des URL, du nom du serveur, du port du serveur, du nom de l'utilisateur authentifié, etc. Enfin, la dernière catégorie s'appelle `environ`. Le tableau ci-dessous donne les informations les plus utiles.

Table 1 : Exemple de configurations matérielles et logicielles

Nom	Description
`PATH_INFO`	L'URI demandée
`HTTP_HOST`	L'adresse serveur
`HTTP_REFERER`	L'URL d'où l'utilisateur vient
`REMOTE_ADDR`	L'adresse du poste client
`HTTP_USER_AGENT`	Le nom et la version du navigateur de l'utilisateur
`REQUEST_METHOD`	La méthode de requête utilisée

Lionel Tricon

Emballez vos sources avec Checkinstall

Soucieux de la bonne santé de votre système d'exploitation, vous n'installez que des logiciels livrés sous forme de paquets et compilés spécifiquement pour votre distribution préférée. Mais que faire le jour où vous vous trouvez confronté à un logiciel qui n'existe qu'au format TGZ ? L'expérience vous a douloureusement démontré que l'installation à partir des sources éparpille inévitablement une multitude de fichiers au sein de votre arborescence disque ; fichiers qu'il devient alors difficile de localiser, puis de supprimer lors d'une tentative de désinstallation ultérieure.

L'utilisation d'un gestionnaire de paquets de type `deb` ou `rpm` offre de nombreux avantages dont l'un, non des moindres, est de garantir l'intégrité de votre base logicielle. Ce qui est gênant, c'est que l'installation d'un logiciel à partir des sources perturbe cette intégrité (les fichiers installés de cette façon ne sont pas connus par le gestionnaire de paquets).

Évidemment, il peut arriver qu'un fichier `Makefile` bien conçu offre une option permettant de désinstaller proprement le programme, mais ce n'est pas systématique. Et que faire lorsque l'on change volontairement le chemin d'installation par défaut du logiciel ? Une désinstallation et une réinstallation à partir des sources plus tard, vous aurez tout bonnement oublié où vous avez installé le susdit logiciel. Vous ne serez même plus en mesure de vous rappeler que celui-ci existe...

C'est justement sur le constat de ces limitations assez pénibles pour celui qui compile et installe régulièrement des logiciels à partir des sources que Felipe Eduardo Sánchez Díaz Durán a pensé et conçu `Checkinstall` **[1]**.

Logiciel assez ancien (la première version date de la fin de l'année 2000) et stable, `Ckeckinstall` propose de générer un paquet de type `deb`, `rpm` ou `slackware` à partir des sources d'un logiciel qui se conformerait de préférence au standard `GNU Autoconf`. Vous n'autorisez plus une installation directe des fichiers sur votre disque, mais une installation a posteriori d'un paquet généré par ce logiciel.

Le fonctionnement est assez simple. `Checkinstall` va surcharger la commande `make install` pour suivre toutes les modifications du système de fichiers à la trace grâce à l'utilitaire `Installwatch` **[2]** écrit par Pancrazio 'Ezio' de Mauro. Ce petit utilitaire bien pratique, qui fonctionne avec tous les binaires de type ELF dynamiquement liés, intercepte tous les appels système qui altèrent le système de fichiers et va permettre à `Ckeckinstall` de générer un paquet installable à partir de la liste des modifications qu'il aura détectées.

La base technique ayant été posée, passons sans plus attendre aux travaux pratiques.

`Checkinstall` est fourni en standard sur la majorité des distributions et intègre l'utilitaire `Installwatch` pour plus de confort à l'installation. Faites donc appel à votre gestionnaire de paquets pour l'installer de la façon la plus habituelle possible. Dans le cas plutôt amusant où celui-ci n'est pas disponible pour votre distribution, n'hésitez pas à l'installer à partir des sources... pour ensuite générer un paquet en s'auto-installant lui-même (une installation quasi-récursive qui séduira les puristes).

Dans le monde Linux, il est usuel d'installer un logiciel à partir des sources avec la séquence de commandes standard `./configure && make && make install`. Un petit rappel sur le sujet n'est peut-être pas inutile.

Petite décomposition du procédé :

- Le script *shell* `configure` se borne à détecter les outils (compilateurs, bibliothèques...) mis à disposition par la chaîne de compilation présente sur votre système (habituellement GNU dans le monde Linux) et permet de vérifier ainsi que tous les outils nécessaires à la bonne compilation du programme sont disponibles dans les bonnes versions ; en cas de succès, le script génère tous les fichiers `Makefile` nécessaires à la bonne compilation du logiciel.
- La commande `make` s'acquitte de la tâche de compiler puis de *linker* les exécutables et les différentes bibliothèques en déroulant le fichier `Makefile` présent à la racine du logiciel ; à ce stade, rien n'est installé sur le disque, tout reste cloisonné dans les sources du logiciel.
- L'appel à la commande `make install` s'applique enfin à la délicate tâche d'installer le logiciel dans votre arborescence disque. Autant les deux phases précédentes ne nécessitent pas de droits particuliers, autant cette dernière étape nécessite de passer temporairement en super-utilisateur pour des raisons évidentes d'accès au système de fichiers.

Prenons un exemple concret en essayant d'installer le logiciel `wget` à partir des sources (les sources sont disponibles sur **[3]**). Au début, nous allons dérouler les deux premières étapes de la façon la plus naturelle possible :

```
$ tar xzvf wget-1.10.2.tar.gz
$ cd wget-1.10.2
$ ./configure
$ make
```

A ce stade, nous n'allons pas faire appel au sempiternel `make install`, mais directement appeler `checkinstall` à la place : le logiciel va alors extraire le maximum d'informations des scripts `Autoconf` (nom, version, licence...) et va vous proposer d'éditer ces champs informatifs qui lui seront nécessaires ultérieurement pour générer le paquet. Il vous sera par exemple demandé de saisir une petite description du paquet logiciel, information qu'il ne peut lui-même déduire des sources.

```
$ /usr/sbin/checkinstall
checkinstall 1.6.0, Copyright 2002 Felipe Eduardo Sanchez Diaz
Duran
          This software is released under the GNU GPL.
The package documentation directory ./doc-pak does not exist.
Should I create a default set of package docs?  [y]: y
Preparing package documentation...OK
Please write a description for the package.
End your description with an empty line or EOF.
>> Aspirateur de sites web
>>
**************************************
**** RPM package creation selected ***
**************************************
This package will be built according to these values:
1 -  Summary: [ Aspirateur de sites web ]
2 -  Name:    [ wget ]
3 -  Version: [ 1.10.2 ]
4 -  Release: [ 1 ]
5 -  License: [ GPL ]
6 -  Group:   [ Applications/System ]
7 -  Architecture: [ i386 ]
8 -  Source location: [ wget-1.10.2 ]
9 -  Alternate source location: [  ]
10 - Requires: [  ]
11 - Provides: [ wget ]
Enter a number to change any of them or press ENTER to continue:
<enter>
Installing with make install...
****************************************************************
 Done. The new package has been saved to
 /usr/src/packages/RPMS/i386/wget-1.10.2-1.i386.rpm
 You can install it in your system anytime using:
      rpm -i wget-1.10.2-1.i386.rpm
****************************************************************
```

Voilà, c'est fait. Fin observateur, vous aurez noté que, par défaut, le paquet généré est de type RPM.

Pour changer ce comportement, vous avez le choix entre deux possibilités : la première consiste à passer une option en ligne de commande lors de l'appel de `checkinstall` pour le forcer à générer un autre type de paquet (par exemple, `-t debian` permet de générer un fichier `.deb` si l'utilitaire `dpkg` est disponible sur votre distribution) ; la seconde consiste à éditer le fichier de configuration `/etc/checkinstallrc` (champ `INSTYPE`).

En ligne de commande, l'option `-h` permet de consulter toutes les options paramétrables lors de l'appel à `checkinstall` ; par symétrie, toutes ces options sont configurables en dur dans le fichier `/etc/checkinstallrc` (les options en ligne de commande priment sur celles du fichier de configuration) sachant que la plus intéressante sera sans nulle doute celle permettant de sélectionner le type de paquets générés par défaut.

Il est aussi possible d'installer directement le paquet en modifiant le champ `INSTALL` du fichier de configuration ou d'appeler l'option `--install=yes` en ligne de commande, mais cela n'est pas conseillé si vous souhaitez déployer par la suite votre paquet sur d'autres distributions. L'étape ultime consiste à installer le paquet sur votre distribution :

```
$ rpm -ivh /usr/src/packages/RPMS/i386/wget-1.10.2-1.i386.rpm
```

L'exemple précédent met en œuvre l'installation d'un paquet RPM. Les utilisateurs de la distribution Debian utiliseront alors la commande `dpkg -i`, alors que ceux de la distribution Slackware utiliseront `installpkg`.

Notez que `checkinstall` a aussi ses limites (même si, personnellement, je n'y ai pas été confronté) et qu'il fonctionnera d'autant mieux que les sources utilisent le *framework* de compilation `GNU Autoconf`. Par contre, dans le cas pas si exceptionnel que ça où le script d'installation est fantaisiste, comme par exemple `install.sh`, il est tout à fait possible de passer le nom du script en option à `checkinstall`.

La séquence sera alors : `./configure && make && checkinstall install.sh`.

Dans l'éventualité d'une installation à partir de sources non conformes au standard `GNU Autoconf`, pas de panique, cela fonctionne parfaitement.

Prenons l'exemple d'un `Makefile` maison qui installe un exécutable `main` et une bibliothèque partagée `lib.so` dans, successivement, les répertoires `/sbin` et `/lib`.

```
all: lib.so main
main: main.c
        g++ -Wall -rdynamic -o main main.c -ldl
lib.so: lib.c
        g++ -Wall -shared -o lib.so lib.c
clean:
        rm -f lib.so main
install:
        cp -f main /sbin
        cp -f lib.so /lib
```

Un appel à `checkinstall` plus tard et on peut constater que le fichier RPM généré remplit parfaitement son office :

```
$ rpm -qpl /usr/src/packages/RPMS/i386/TEST-20070215-
1.i386.rpm
/lib
/lib/lib.so
/sbin
/sbin/main
```

Que dire de plus sur ce petit utilitaire fort sympathique, sinon qu'il sera plus particulièrement utile à ceux qui compilent régulièrement des applications à partir des sources et qui souhaitent éviter par dessus tout de rendre leur système de fichiers incohérent (le syndrome Windows en somme)? C'est en outre un outil indispensable pour ceux qui administrent des machines serveurs et qui déploient un logiciel à l'identique sur plusieurs systèmes à la fois.

Pour toutes ces raisons, pour sa capacité à générer des paquets `Debian`, `Redhat` ou `Slackware`, et pour son interface assez pratique permettant de modifier sans douleur les champs d'informations associés au paquet, ce petit logiciel devrait assez rapidement tenir une place de choix dans votre logithèque libre.

Bons emballages de sources !

Lionel Tricon,

lionel.tricon@free.fr

LIENS

- [1] Checkinstall : http://asic-linux.com.mx/~izto/checkinstall/
- [2] Installwatch : http://freshmeat.net/projects/installwatch/
- [3] Télécharger wget : http://ftp.gnu.org/pub/gnu/wget/

GLMF, le magazine qui code plus l33t que la musique, présente :

Yann Guidon

Algorithmique/C/Optimisation : signature ultrarapide de blocs de données (la saga des CRC continue)

Dans l'article de décembre 2005 (GLMF n°78) sur la conception d'une routine de CRC 16 bits rapide, je croyais sincèrement avoir atteint les limites théoriques et pratiques du domaine. Mais un morceau de code n'est jamais le plus rapide dans l'absolu ! En appliquant des techniques utilisées par les générateurs de nombres pseudo-aléatoires, exposées (il n'y a pas de hasard) dans le numéro de mars 2006, nous pouvons encore gagner un ordre de grandeur de vitesse. Nous y perdons la compatibilité avec l'algorithme CRC16 initialement choisi, mais cela débloque la sixième vitesse de nos microprocesseurs. Attachez bien votre ceinture...

1. Résumé des épisodes précédents

Cela fait maintenant plus d'un an que cette histoire de CRC a commencé, une *petite digression* dans la série sur la compression des données. Pour les lecteurs qui (re)prennent le train en marche, je remets ici tout en perspective :

* Décembre 2005, GLMF n°78 :
Description sommaire d'un algorithme de CRC classique, choix et validation des paramètres pour signer sur 16 bits et à haute vitesse, des blocs de données de 4K octets ou moins. La problématique est posée mais la solution trouvée, malgré les optimisations poussées, n'est *pas vraiment foudroyante*.

* Mars 2006, GLMF n°81 :
Découverte des Corps de Galois, de la multiplication modulaire, de `GF(2)` et des LFSR. Cet article assez mathématique a des applications directes et concrètes en informatique, en particulier pour les CRC, dont les LFSR sont justement très proches. Un générateur de nombres pseudo-aléatoires (plus précisément, un *tGFSR* de 4 mots brouillé par un CRC32) a été conçu et testé. En raison de son efficacité, l'idée de le transformer en CRC germe, mais la mutation n'est pas tout à fait évidente : il reste des problèmes théoriques et algorithmiques, que je me suis efforcé de résoudre depuis.

* Juillet 2006, GLMF n°85 :
Une vue unifiée des registres à décalage (CRC et LFSR) est proposée, débouchant sur un autre type de CRC : le pseudo-CRC. Celui-ci conserve presque toutes les propriétés d'un CRC classique, mais simplifie un petit peu la conception des programmes. L'importance des polynômes générateurs est aussi examinée, confortant le choix du polynôme `0x8005`. D'autres aspects théoriques sont abordés et vérifiés pour préparer sereinement le présent article.

* Février 2007, GLMF n°91 :
Quelques techniques de codage en langage C sont exposées, et utilisées ici.

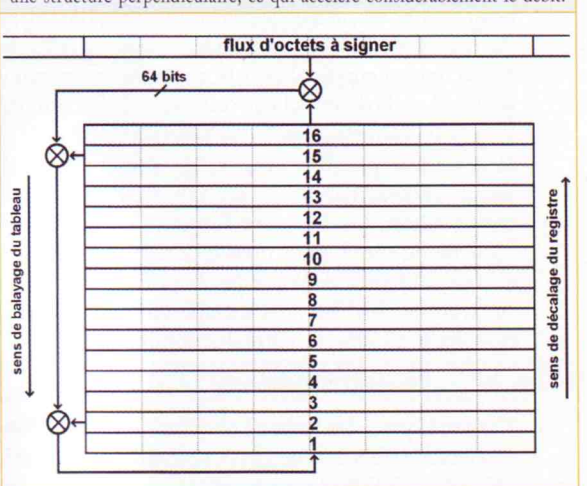
L'application directe du pseudo-CRC, décrit dans GLMF n°85, est une routine à très haute performance, telle qu'imaginée initialement à la suite de GLMF n°78. L'aspect le plus important de cet article (en dehors des nombreuses digressions sur le codage MMX et quelques nuances mathématiques) est la mise au point des techniques de correction de l'alignement des blocs et de l'*Endian* du processeur, qui formeront la base pour d'autres algorithmes. On peut considérer cet exposé comme un grand banc d'essai qui permet de défricher sereinement les difficultés, afin de réaliser des versions plus sophistiquées et solides dans les prochains numéros (si, si, c'est possible).

Puisque j'ai fait de mon mieux pour éclairer les zones sombres dans les numéros précédents, je ne reviendrai pas dessus ici. Il est donc fortement recommandé de (re)lire les épisodes précédents si quelque chose vous échappe. Ou alors, vous pouvez simplement recopier les codes sources en me faisant confiance, avec les inconnues que cela implique. En tout cas, je pense avoir blindé la théorie, alors passons maintenant à la pratique !

2. Vive le parallélisme !

Développons (enfin) l'idée initiée dans l'article de mars 2006 : le meilleur moyen d'augmenter drastiquement la quantité de calculs, par rapport à une approche conventionnelle, est de paralléliser les registres à décalage et d'y accéder de manière « perpendiculaire ».

Figure 1 : Beaucoup de calculs de CRC peuvent être parallélisés avec une structure perpendiculaire, ce qui accélère considérablement le débit.



Cette méthode, bien pratique pour générer des nombres pseudo-aléatoires, est encore plus intéressante quand elle est appliquée à la signature de blocs de données. De plus, comme les registres parallèles ne communiquent pas entre eux, nous sommes affranchis des deux problèmes majeurs qui compliquent les algorithmes de traitement de blocs :

- Nos programmes ne garantissent pas l'alignement des débuts de blocs, ce qui oblige à ajouter un prologue et un épilogue à la boucle principale, doublant ou triplant ainsi la taille et la complexité du code (comme en décembre 2005). Si les bits du calcul sont indépendants entre eux, on peut probablement simplifier le prologue et l'épilogue.
- L'ordre des octets (« *endianness* ») n'est pas le même pour tous les processeurs, ce qui complique les accès à la mémoire. Les algorithmes optimisés de l'article de décembre 2005 ne peuvent pas être compilés pour des ordinateurs *Big Endian* à base de SPARC, 68k, PPC, PA-RISC et certaines versions de MIPS, SH et ARM (car, pour ne rien simplifier, ces derniers peuvent aussi être configurés en *Little Endian*). Ils subissent une pénalité de vitesse qu'il est difficile de compenser de manière simple et portable. Cependant, si les bits du calcul sont indépendants entre eux, ils sont insensibles à l'ordre des octets et il n'y a besoin de créer qu'une seule version de la fonction, qui fonctionnera au mieux sur toutes les plateformes.

Ou du moins, c'est l'intention.

Afin de maximiser la vitesse, je choisis d'utiliser des mots de 64 bits. En effet, plus on traite d'octets par instruction, plus l'accélération est forte par rapport au code « octet par octet » développé en décembre 2005. Les 16 mots occupent 128 octets en mémoire, soit 4 fois moins que les 512 octets de l'algorithme initial. Tout cela semble prometteur.

2.1 L'algorithme central

Examinons maintenant le cœur de l'algorithme, qui est la boucle principale effectuant le calcul. Cela déterminera ensuite la conception du code de prologue et de l'épilogue. Le fonctionnement du code est illustré par la figure 1.

Tout d'abord, rappelons que l'algorithme est basé sur un LFSR (registre à décalage à rétroaction linéaire) de 16 bits, avec un polynôme générateur égal à `0x8005`.

```
/*
 Ebauche de la boucle principale de signature.
 paramètres :
   U64 *pointeur : pointeur du début du bloc
   U32 taille_du_bloc : taille du bloc en octets
*/
U16 aligned_CRC16_64(U32 taille_du_bloc, U64 *pointeur) {
  U64 CRC16_64_table[16], tmp;
  U32 taille;
  int index=15; /* modulo 16,
 on commence par la fin pour suivre la figure 1 */
  taille=taille_du_bloc >> 3; /* 8 octets à la fois */
  while (taille--) {
    tmp=CRC16_64_table[index];
    tmp^= *(pointeur++);
    tmp^=CRC16_64_table[(index+2)&15];
    tmp^=CRC16_64_table[(index+15)&15];
    CRC16_64_table[index]=tmp;
    index=(index-1) & 15; /* rotation des registres */
  }
  return 0 /* pas encore de compaction */
}
```

La fonction essentielle du code est simple : effectuer un XOR entre les données à signer et les entrées 16, 15 et 2 du tableau, pour placer le résultat à l'entrée 0. L'entrée 0 et l'entrée 16 sont confondues, en raison de la taille et de l'accès cyclique. Et pour éviter de déplacer toutes les entrées, on recalcule les indices des entrées.

Toutefois, par rapport au code publié en mars 2006, le pointeur est déplacé dans le sens inverse. Une erreur d'inattention s'était en effet glissée. Par chance, ce n'est pas grave puisque, grâce à la propriété de symétrie des polynômes générateurs, nous somme retombés sur un cas qui fonctionne aussi parfaitement. Le code présenté en mars 2006 est donc tout à fait valide, mais il est exceptionnel que le hasard fasse si bien les choses...

Mais pour garder le code facile à coder et à comprendre, l'incrémentation de `index` est préférable à la décrémentation : on verra plus tard que le calcul de l'index est plus cohérent. Pour conserver le polynôme, il faut alors inverser les termes, ce qui conduit à utiliser les entrées 14 et 1 au lieu de 15 et 2. C'est ce que fait le code suivant, avec les changements mis en évidence :

```
/* Version modifiée avec balayage dans le sens inverse */
int index=0; /* on commence à 0 grâce à l'inversion. */
taille=taille_du_bloc >> 3;
while (taille--) {
  tmp=CRC16_64_table[index];
  tmp^= *(pointeur++);
  tmp^=CRC16_64_table[(index+1)&15];
  tmp^=CRC16_64_table[(index+14)&15];
  CRC16_64_table[index]=tmp;
  index=(index+1) & 15;
   /* rotation des registres dans l'autre sens */
}
```

L'importance d'un *polynôme clairsemé* (avec peu de termes, voir la définition dans GLMF n°85 de juillet 2006) apparaît immédiatement : cela réduit le nombre d'opérations dans la partie critique du code. Imaginez la lourdeur du code si on avait choisi `0xCBB7` ! Mais pour l'instant, ce qui consomme le plus de temps est l'accès au tableau, car les calculs de pointeurs infligent d'importantes pénalités (c'est l'une des conclusions de l'article de décembre 2005 et on peut le vérifier en examinant le code assembleur généré).

2.2 Optimisation pour les processeurs x86

Le code précédent fonctionne correctement, mais n'est pas particulièrement foudroyant... Si on le compile sur les processeurs x86 classiques (tels Pentium II, Pentium III, Pentium IV, Athlon Thunderbird...), le

compilateur ne va pas utiliser automatiquement les 8 registres de 64 bits disponibles avec les extensions MMX et/ou SSE. La perte est considérable, car, d'une part, le compilateur utilisera deux registres et deux instructions pour traiter une donnée, et, d'autre part (la conséquence directe), il y aura deux fois moins de registres disponibles, ce qui augmente les accès à la mémoire.

Heureusement, depuis GCC3, certaines extensions sont traitées nativement (même si elles ne sont pas automatiques) et il n'est plus nécessaire d'insérer des `__asm__ __volatile__ ();` partout. Cela rend le code plus lisible, portable et maintenable. Je suis bien plus à l'aise lorsque j'utilise directement `nasm`, mais j'ai réussi à écrire une petite couche d'abstraction très simple :

```
/*
 fichier def64.h : Définit les opérations à utiliser
     pour manipuler des données 64 bits avec GCC 3.x
  créé par Yann GUIDON le 16 mai 2006

  Options de compilation conseillées :
   gcc -Os -W -Wall -momit-leaf-frame-pointer \
   -fomit-frame-pointer -march=pentium3 nomdufichier.c
*/
#define U8  unsigned char
#define U16 unsigned short int
#ifndef U32
 #if defined (__alpha)
  #define U32 unsigned int
 #else
  #define U32 unsigned long int
 #endif
#endif
#define U64 unsigned long long int
#define PTR_CAST (long)

#if defined (__MMX__)
  #define U64_XOR(x,y)   __builtin_ia32_pxor(x,y)
  #define U64_OR(x,y)    __builtin_ia32_por(x,y)
  #define U64_AND(x,y)   __builtin_ia32_pand(x,y)
  #define U64_ANDN(x,y)  __builtin_ia32_pandn(x,y)
  #define U64_SHR(x,y)   __builtin_ia32_psrlq(x,y)
  #define U64_SHL(x,y)   __builtin_ia32_psllq(x,y)
#else
  #define U64_XOR(x,y)   (x ^ y)
  #define U64_OR(x,y)    (x | y)
  #define U64_AND(x,y)   (x & y)
  #define U64_ANDN(x,y)  (~x & y)
  #define U64_SHR(x,y)   (x >> y)
  #define U64_SHL(x,y)   (x << y)
#endif
```

NOTE

Les petits caprices de GCC

Les instructions de décalage MMX, en particulier `psrlq` et `psllq` que nous utilisons ici, acceptent deux types d'opérande pour indiquer le nombre de bits à décaler : soit une constante, soit un autre registre MMX.

Cependant, l'implémentation de `__builtin_ia32_psrlq()` et `__builtin_ia32_psllq()` dans `gcc3.3.4` n'accepte pas les arguments constants (pour une raison qui m'échappe totalement, la compilation échoue avec une jolie erreur interne).

La situation a peut-être été corrigée dans les versions suivantes de gcc mais je ne l'ai pas vérifié. C'est en tout cas compensé par la capacité d'optimisation de `gcc`, car je suis tombé par hasard sur le morceau de code suivant :

```
/* code C original */
  int i=42;
  X = U64_SHL(X, i);
; code de décalage généré par gcc3.3.4 pour gas :
   opération optimisée par inférence de valeur
 psllq $42, %mm1
```

Ainsi, si `gcc` détecte que la valeur de `i` est une constante (qui ne sera pas modifiée entre l'affectation à `i` et l'utilisation par `psllq`), il va utiliser la forme *immédiate* de l'instruction et marquer directement la bonne valeur.

On peut aussi remarquer que le même `gcc3.3.4` accepte sans broncher un argument entier normal (`U32`). Quand cet argument est variable, par exemple le compteur d'une boucle, le compilateur va émettre la suite suivante d'instructions :

```
/* code C original */
  int i;
  for (i=0; i<n; i++) {
    ....
    X = U64_SHL(X, i);
    ....
  }
; code de décalage généré par gcc3.3.4 pour gas :
 movl %eax, (%esp)  ; eax est l'entier, mis sur la pile
 movl %eax, %ecx    ; recopie l'entier
 sarl $31, ecx      ; ecx contient le signe de eax
 movl %ecx, 4(%esp) ; met aussi le signe sur la pile
 movq (%esp), %mm0  ; lit l'ensemble du mot d'un coup
 psllq %mm0, %mm1   ; effectue (enfin) le décalage
```

D'abord, la transmission de la valeur de `i` sur la pile est une stratégie qui m'échappe, car il y a toujours le risque que `esp` ne soit pas aligné sur 8 octets, ce qui risque d'infliger une pénalité de quelques cycles à l'exécution. Il y a aussi toutes ces manipulations pour obtenir le signe de `eax`, qui occupe 3 instructions alors qu'au final, l'instruction `psllq` ne lit que les 6 bits de poids faible. Une simple instruction `movd %eax, %mm0` aurait remplacé les cinq premières instructions.

Une solution est de conserver la valeur non pas dans un entier normal, mais dans un des registres MMX. La technique de forçage est expliquée à la fin de la partie **3.4**. Une autre mesure à essayer est d'utiliser des entiers **non signés**, tels `U32`, pour éviter l'extension du signe. Enfin, la simplicité des décalages par un nombre immédiat est une raison supplémentaire de dérouler les boucles.

Toutefois, ce n'est pas toujours facile à coder et le compilateur va toujours vouloir passer par la pile pour transférer la valeur... Afin de faciliter l'écriture de certaines parties du code qui va être développé plus loin dans cet article, j'ai donc ajouté ces définitions :

```
/* Correction du fichier def64.h */
#if defined (__MMX__)

/* 3 cas particuliers pour le décalage : */

/* - Si y est un registre MMX : tout baigne */
  #define U64_SHR(x,y)   __builtin_ia32_psrlq(x,y)
  #define U64_SHL(x,y)   __builtin_ia32_psllq(x,y)

/* - Si y est un entier constant : il faut ruser */
  #define U64_SHRi(x,y) ({       \
   unsigned tmp=y;               \
   __builtin_ia32_psrlq(x,tmp); })
  #define U64_SHLi(x,y) ({       \
   unsigned tmp=y;               \
   __builtin_ia32_psllq(x,tmp); })

/* - Si y est un registre normal : la tuile !
Comme movd est indisponible
dans les intrinsèques, il faut le définir. */
#define U64_MOVD(x,y)           \
   __asm__ __volatile__ ( "\t"  \
     "movd %1, %0\n\t"          \
   : "=X" (x)                   \
   : "r" (y) );

#define U64_SHLr(x, y) ({\
   U64 r;                \
   U64_MOVD(r,y);        \
   U64_SHL(x,r);        })

#define U64_SHRr(x, y) ({\
   U64 r;                \
   U64_MOVD(r,y);        \
   U64_SHR(x,r);        })

#else
  #define U64_SHR(x,y)   (x >> y)
  #define U64_SHL(x,y)   (x << y)
  #define U64_SHRi(x,y)  U64_SHR(x,y)
  #define U64_SHLi(x,y)  U64_SHL(x,y)
  #define U64_SHRr(x,y)  U64_SHR(x,y)
  #define U64_SHLr(x,y)  U64_SHL(x,y)
#endif
```

Après bien des efforts, j'ai pu rendre tout cela fonctionnel et flexible, pour éviter au maximum les `#if defined (__MMX__)` dans les codes sources. Le plus ennuyeux est l'absence de `movd` et `movq` dans les intrinsèques, ce qui oblige à de nombreuses contorsions syntaxiques pas toujours portables.

Moralité : ce n'est pas pour rien que la première recommandation (probablement la plus importante) donnée dans GLMF n°91 est de **toujours examiner la sortie du compilateur !** Sans ces « raffinements », le code généré par les programmes que nous allons développer serait encombré de références inutiles à `esp`.

Ces définitions sont d'une grande simplicité à utiliser, en particulier puisque les `__builtin_ia32_XXXX()` retournent directement le résultat, comme une fonction ou une expression, tout en *émettant* l'opcode correspondant. C'est intéressant dans notre cas, puisque nous pouvons écrire le XOR des quatre valeurs de manière imbriquée, ce qui laisse plus de liberté au compilateur pour allouer les registres et (éventuellement) ordonnancer les instructions.

```
U64 CRC16_64_table[16]; /* en mémoire globale
                           pour simplifier l'adressage */
```

```
/* Paramètres :
  taille du bloc en octets et pointeur du début du bloc */
void aligned_CRC16_64(U32 taille_du_bloc, U64 *pointeur) {
  int taille, index; /* modulo 16 */

  /* initialisation */
  memcpy(CRC16_64_table, CRC16_64_const,
                    sizeof(CRC16_64_const));
  index=0;

  taille=taille_du_bloc >> 3;
  while (taille--) {
    CRC16_64_table[index] =
      U64_XOR(
        U64_XOR(
          CRC16_64_table[index],
          *(pointeur++)),
        U64_XOR(
          CRC16_64_table[(index+1) &15],
          CRC16_64_table[(index+14)&15] ));
    index=(index+1) & 15; /* rotation des registres */
  }
}
```

Après compilation, la taille de la boucle a presque diminué de moitié grâce aux nouvelles instructions :

```
.L6:
        movq    CRC16_64_table(,%esi,8), %mm1
        leal    1(%esi), %edx
        andl    $15, %edx
        movq    CRC16_64_table(,%edx,8), %mm0
        leal    14(%esi), %eax
        andl    $15, %eax
        pxor    (%ebx), %mm1
        decl    %ecx
        addl    $8, %ebx
        pxor    CRC16_64_table(,%eax,8), %mm0
        pxor    %mm0, %mm1
        movq    %mm1, CRC16_64_table(,%esi,8)
        incl    %esi
        andl    $15, %esi
        cmpl    $-1, %ecx
        jne     .L6
```

Les instructions en rouge remplacent deux opérations standard sur 32 bits. Toutefois, l'adressage circulaire du tableau gaspille toujours six instructions et il reste encore des mouvements de données redondants entre la mémoire et les registres.

2.3 Petite vérification préliminaire

Notre contrainte initiale, exposée dans GLMF n°78 de décembre 2005, prenait en compte des blocs de données d'une taille de 4096 octets au maximum. Cette limite est justifiée par les 16 bits de la signature, qui reboucle au bout de 32767 bits signés dans les cas défavorables. Nous comptons un facteur 2, car la taille du trinôme générateur principal de `0x8005` est sur 15 bits (GLMF n°85 montre que `0x(1)8005=0x3×0x8003` dans `GF(2)`).

Maintenant, la configuration des registres a un peu changé. Il n'y en a plus un seul, mais 64 en parallèle. Sachant qu'un registre (de poly `0x8005`, en mode LFSR) reboucle au bout de 65534 itérations, combien d'octets identiques peut-on signer avant de faire reboucler les 128 octets ? Encore une fois, un petit programme nous permet de le mesurer.

```
/* la fonction aligned_CRC16_64() est modifiée pour ne pas
incrémenter le pointeur, afin de lire constamment 0 */
        ...
/* code de comptage dans le main() : */
  U64 buffer_0=0; /* donnée lue en boucle */
  int j=0;
  do {
    j+=8;
    aligned_CRC16_64(j,&buffer_0);
  } while (memcmp(CRC16_64_table,CRC16_64_const, 8*16));
  printf("j=%d\n",j);
        ...
#./a.out
j=4194176  (après 40 minutes car c'est un algo en O(n^2))
```

On remarque d'abord que la période n'est pas nulle, même lorsque l'algorithme est alimenté de zéros. L'opus de juillet 2006 (GLMF n°85) a exploré les défauts du pseudo-CRC, dont le registre n'est pas mis à jour si l'entrée et le registre sont nuls. Or `CRC16_64_table` est initialisé ici avec la table (non nulle) décrite dans la **partie 3.1**. Le pseudo-CRC est correctement amorcé, et tourne donc bien en « mode LFSR ».

L'algorithme reboucle au bout de *presque* 4MiOctets (ou 32Mi bits). Ce nombre s'écrit aussi `0x3FFF80` et est composé par 64×65534. On peut identifier 65534 comme la période du LFSR correspondant, et 64 serait le nombre de registres en parallèle.

Seulement, 64×65534 donne ici la période en octets et non en bits. Le facteur 8 manquant est lié à la rotation des registres, puisque dans un LFSR ou CRC normal, en raison de la paternité avec la multiplication modulaire, le registre est décalé à chaque itération. Or, **notre code n'effectue pas de rotation, il compare les registres non décalés.**

Pour retrouver le facteur 8 manquant, il faut considérer une période de rebouclage de 65534 itérations. Un rebouclage correspond à une copie de la table des registres, après un décalage de (65534 mod 16)=14, ou 2 si on regarde dans l'autre sens.

```
Etat d'un des registres de CRC à l'itération i :
 A B C D E F G H I J K L M N O P
Etat du registre à l'itération i+65534 :
 C D E F G H I J K L M N O P A B
 |<------------------------>|
 déphasage de 65534 mod 16 = 14 positions
```

Puisque nous avons 16 positions, il faut 16/2=8 rebouclages pour retrouver la situation initiale. Notre période est donc bien de **8×64×65534 bits**.

EXEMPLE

Illustration du déphasage avec un LFSR de 4 bits :

Reprenons le LFSR de 4 bits en configuration de Galois présenté dans le GLMF n°81 de mars 2006. Ce LFSR a une période de (2^4)-1=15 itérations, et son registre passe par les états

`2, 4, 8, 3, 6, C, B, 5, A, 7, E, F, D, 9, 1`

(en hexa) avant de reboucler.

Question : Combien de nombres va-t-il générer avant de reboucler si on en tire plusieurs valeurs avant de comparer la dernière avec la valeur initiale ?

La réponse dépend du nombre de valeurs extraites. Dans cet exemple, le LFSR est initialisé à 1, puis son état est vérifié tous les 4 calculs :

```
                                                1
(2  4  8  3)(6  C  B  5)(A  7  E  F)(D  9  1  2)
(4  8  3  6)(C  B  5  A)(7  E  F  D)(9  1  2  4)
(8  3  6  C)(B  5  A  7)(E  F  D  9)(1  2  4  8)
(3  6  C  B)(5  A  7  E)(F  D  9  1)
```

Total : 60 nombres émis, soit 15 (période du LFSR) × 4 (taille du bloc), chaque nombre apparaît 4 fois.

Le placement des nombres permet de visualiser le cycle original, qui se décale d'une position au fur et à mesure. On peut remarquer un fait intéressant pour nous : bien que les nombres tirés se répètent selon la période naturelle du LFSR, les nombres entre parenthèses forment un groupe qui n'est pas répété.

Ainsi, si on génère plusieurs itérations d'un coup, comme dans le LFSR du GLMF n°78, il est possible d'obtenir plus de bits avant rebouclage. Le LFSR 32 bits de décembre 2005 a une période intrinsèque de (2^32)-1 itérations, mais génère 4 bits à la fois. Donc, la séquence de sortie reboucle au bout de 4×((2^32)-1) bits ou (2^32)-1 appels, soit 2GiOctets-4bits. Si on extrait 8 bits à la fois (avec 2 appels consécutifs au générateur), on passe à 4GiOctets-1. Ce petit « -1 » permet donc de « gonfler » la période artificiellement, si on considère des symboles de plus grande taille.

Dans le cas qui nous concerne, la période du poly `0x8005` n'est pas de (2^16)-1, mais (2^16)-2 itérations (ou plutôt 2×((2^15)-1), car le polynôme est composé). Le décalage est de 2 positions, il faut donc deux fois moins d'itérations pour que le registre reboucle complétement.

NOTE

On peut donc conclure, à ce niveau, que notre approche *optimisée* (qui économise les décalages), bien que mathématiquement légèrement incorrecte, est en fait meilleure, puisqu'elle fait aussi gagner un facteur 8 sur la longueur maximale des blocs testables. Cet argument est important pour plus tard, car l'absence de rotation simplifiera aussi beaucoup la *compaction*, qui sera donc plus rapide.

Enfin, puisque nous retrouvons des valeurs cohérentes avec les expérimentations réalisées dans les articles précédents, l'algorithme de référence est validé.

2.4 L'épilogue

La fonction `aligned_CRC16_64()` servira de référence pour la conception des autres versions (encore plus optimisées) de l'algorithme de signature. Les signatures générées par toutes les versions dérivées seront comparées pour s'assurer que leur fonctionnement est correct. Mais, en plus de ne pouvoir signer que des blocs alignés, la version actuelle ne supporte pas les blocs de taille arbitraire. Ce dernier point risque de rendre les vérifications ultérieures difficiles, nous allons donc nous en occuper tout de suite.

Permettre les tailles non multiples de 8 est assez aisé : il suffit de calculer un mot supplémentaire, mais de ne tenir compte que des octets qui nous intéressent. Pour cela, il nous faut juste un bon masque et quelques opérations booléennes :

```
/* épilogue */
  taille=taille_du_bloc & 7;  /* garde juste les 3 LSB */
  if (taille) {
    /* calcule un dernier élément */
    t = CRC16_64_table[index];
    u = U64_XOR(
      U64_XOR(t, *pointeur),
      U64_XOR(CRC16_64_table[(index+1) &15],
              CRC16_64_table[(index+14)&15] ));
    /* construction du masque */
    mask = (U64)-1LL;
    taille <<= 3;
    mask = U64_SHLr(mask, taille);  /* LittleEndian ! */
    /* sélectionne les bons octets */
    t    = U64_AND(t,mask);
    mask = U64_ANDN(mask, u);
    t    = U64_OR(mask, t) ;
    CRC16_64_table[index] = t;
  }
```

On remarque que le masque dépend de l'ordre des octets dans la machine. Ici, le code est valable pour une machine Little Endian, il faudra inverser le décalage pour une machine Big Endian. J'ai donc ajouté les définitions suivantes au fichier `def64.h` :

```
/* Ajout au fichier def64.h */

/* Adaptation du sens de décalage */
#if __BYTE_ORDER == __LITTLE_ENDIAN
# define LOCAL_SHL(x,y)  U64_SHL(x,y)
# define LOCAL_SHR(x,y)  U64_SHR(x,y)
# define LOCAL_SHLi(x,y) U64_SHLi(x,y)
# define LOCAL_SHRi(x,y) U64_SHRi(x,y)
# define LOCAL_SHLr(x,y) U64_SHLr(x,y)
# define LOCAL_SHRr(x,y) U64_SHRr(x,y)
# define CONVERT_LE64(x) (x)
#else
# if __BYTE_ORDER == __BIG_ENDIAN
#  define LOCAL_SHL(x,y)  U64_SHR(x,y)
#  define LOCAL_SHR(x,y)  U64_SHL(x,y)
#  define LOCAL_SHLi(x,y) U64_SHRi(x,y)
#  define LOCAL_SHRi(x,y) U64_SHLi(x,y)
#  define LOCAL_SHLr(x,y) U64_SHRr(x,y)
#  define LOCAL_SHRr(x,y) U64_SHLr(x,y)
/* adapté du linux/byteorder/swab.h de Faré : */

#  define CONVERT_LE64(x) ({ U64 __x = (x); \
 ((U64)( \
  (U64)(((U64)(__x)&(U64)0x00000000000000ffULL)<<56)|  \
  (U64)(((U64)(__x)&(U64)0x000000000000ff00ULL)<<40)|  \
  (U64)(((U64)(__x)&(U64)0x0000000000ff0000ULL)<<24)|  \
  (U64)(((U64)(__x)&(U64)0x00000000ff000000ULL)<< 8)|  \
  (U64)(((U64)(__x)&(U64)0x000000ff00000000ULL)>> 8)|  \
  (U64)(((U64)(__x)&(U64)0x0000ff0000000000ULL)>>24)|  \
  (U64)(((U64)(__x)&(U64)0x00ff000000000000ULL)>>40)|  \
  (U64)(((U64)(__x)&(U64)0xff00000000000000ULL)>>56)));\
})
# else
#  error __BYTE_ORDER non supporté ou indéfini !
# endif
#endif
```

Toutes ces définitions seront utilisées par la suite. En attendant, dans le morceau de code source précédent, il faut remplacer

```
    mask = U64_SHLr(mask, taille);
```

par

```
    mask = LOCAL_SHLr(mask, taille);
```

2.5 Optimisation en « pipeline »

Pour accélérer encore le code, voici une technique qui repose sur le fait que les termes du polynôme sont regroupés. Je l'ai nommée « pipeline », car l'idée est similaire à une chaîne de montage de voitures, où *les opérations sont organisées autour des tâches à effectuer*, au lieu des structures à traiter. On va garder dans des variables temporaires (donc dans des registres du processeur) les entrées du tableau correspondant aux termes à utiliser immédiatement pour le calcul, ce qui ne prend que quatre registres.

En effet, notre cher polynôme `0x8005` possède l'heureuse propriété d'avoir un grand « trou », puisque les bits 2 à 13 sont inutilisés. On n'a donc besoin à un moment donné que de 3 des 4 entrées consécutives et les 4 registres temporaires nécessaires tiennent facilement dans les 8 registres disponibles en MMX.

De plus, 4 est un sous-multiple de la taille du registre à décalage (à 16 entrées), ce qui facilite énormément le déroulage d'une boucle, donc l'économie d'un index.

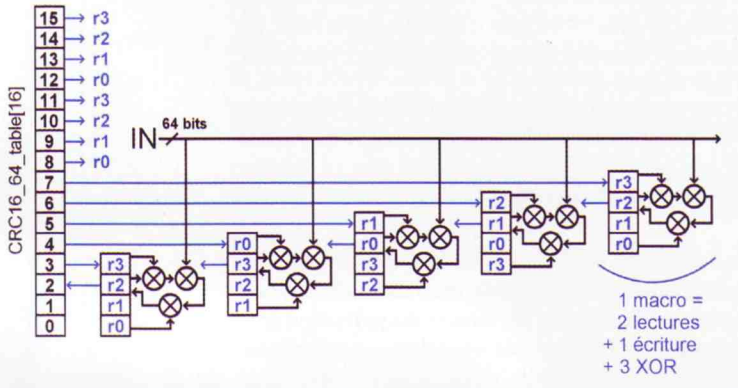
Pour comprendre un peu mieux comment programmer cet algorithme peu courant, regardons les accès successifs au tableau. La liste suivante a été générée par `aligned_CRC16_64()`, instrumenté au moyen de quelques `printf()` :

```
[ 0]^=*p^[ 1]^[14]   - 0,1,2
[ 1]^=*p^[ 2]^[15]   - 1,2,3
[ 2]^=*p^[ 3]^[ 0]   - 2,3,0
[ 3]^=*p^[ 4]^[ 1]   - 3,0,1
[ 4]^=*p^[ 5]^[ 2]   - 0,1,2
[ 5]^=*p^[ 6]^[ 3]   - 1,2,3
[ 6]^=*p^[ 7]^[ 4]   - 2,3,0
[ 7]^=*p^[ 8]^[ 5]   - 3,0,1
[ 8]^=*p^[ 9]^[ 6]   - 0,1,2
[ 9]^=*p^[10]^[ 7]   - 1,2,3
[10]^=*p^[11]^[ 8]   - 2,3,0
[11]^=*p^[12]^[ 9]   - 3,0,1
[12]^=*p^[13]^[10]   - 0,1,2
[13]^=*p^[14]^[11]   - 1,2,3
```

```
[14]^=*p^[15]^[12]   - 2,3,Ø
[15]^=*p^[ Ø]^[13]   - 3,Ø,1
[ Ø]^=*p^[ 1]^[14]   - Ø,1,2
... et ça reboucle
```

La partie de gauche indique les indices utilisés pour chaque étape du calcul, et la colonne de droite contient ces mêmes indices *modulo 4*. On peut observer qu'il n'y a pas de recouvrement sur les indices de la colonne de droite. Ils vont donc nous servir à nommer les registres temporaires.

Figure 2 : Transformation du pseudo-CRC pour réduire le nombre de registres de travail



En partant du code initial ainsi que des informations précédentes, j'ai construit le système de macros ci-dessous. Malheureusement, `cpp` n'est pas coopératif pour effectuer les calculs de modulo. J'ai donc dû calculer les numéros des registres à la main. Je pourrais utiliser `m4`, mais le processus de compilation serait inutilement plus complexe.

```
/* aligned_CRC16_64_2 : fonction de CRC16-64 déroulée.
  Elle travaille sur des blocs de 16 mots de 64 bits
  (128 octets à la fois par itération),
  pointeur aligné sur 8 octets. */
U16 aligned_CRC16_64_2(U32 taille_du_bloc, U64 *p) {
  int taille = taille_du_bloc >> 7;
  U64 rØ, r1, r2, r3;
  /* initialisation */
  memcpy(CRC16_64_table,CRC16_64_const,8*16);
  rØ = CRC16_64_table[Ø];
/*r1 = CRC16_64_table[1];
    effectué lors de la première itération */
  r2 = CRC16_64_table[14];
  r3 = CRC16_64_table[15];
  #define GFSR_ROUND(x, a, b, c)              \
      r##b = CRC16_64_table[(x+1) & 15];      \
      r##a = U64_XOR( U64_XOR( r##a , p[x]),  \
                      U64_XOR( r##b , r##c )); \
      CRC16_64_table[(x) & 15] = r##a ;
    while (taille--) { /* l'instanciation :
                  a = (x) % 4
                  |  b = (x+1) % 4
                  |  |  c = (x+14) % 4  */
    GFSR_ROUND( Ø, Ø, 1, 2)
    GFSR_ROUND( 1, 1, 2, 3)
    GFSR_ROUND( 2, 2, 3, Ø)
    GFSR_ROUND( 3, 3, Ø, 1)
    GFSR_ROUND( 4, Ø, 1, 2)
    GFSR_ROUND( 5, 1, 2, 3)
    GFSR_ROUND( 6, 2, 3, Ø)
    GFSR_ROUND( 7, 3, Ø, 1)
    GFSR_ROUND( 8, Ø, 1, 2)
    GFSR_ROUND( 9, 1, 2, 3)
    GFSR_ROUND(1Ø, 2, 3, Ø)
    GFSR_ROUND(11, 3, Ø, 1)
    GFSR_ROUND(12, Ø, 1, 2)
    GFSR_ROUND(13, 1, 2, 3)
    GFSR_ROUND(14, 2, 3, Ø)
    GFSR_ROUND(15, 3, Ø, 1)
    p+=16;
  }
  return Ø; /* pas encore de compaction */
}
```

Le code généré par `GCC 3.3.4` ne contient que 5 instructions seulement pour chaque instanciation de la macro, avec des modes d'adressage très simples donc rapides :

```
movq %mm2, CRC16_64_table+16 ; sauve le résultat précédent
pxor 24(%eax), %mm1; combinaison avec la donnée à signer
pxor %mm3, %mmØ    ; combinaison avec les autres registres
pxor %mmØ, %mm1
movq CRC16_64_table+4Ø, %mmØ ; lecture prochaine opérande
```

Je pense que ce code nécessite environ trois à six cycles de processeur par itération, tout en traitant huit octets simultanément, ce qui est considérablement plus rapide que les cinq à huit cycles *par octet* pour les codes des CRC16 publiés en décembre 2005. L'accélération est environ d'un facteur dix !

2.6 Optimisations plus poussées

Les derniers facteurs de ralentissement de ce code sont simples :

- D'abord, deux accès à la mémoire pour chaque itération sont inutiles et devraient pouvoir être évités, si au moins il y avait suffisamment de registres. Une boucle déroulée occupant 16 registres serait possible sur MIPS-64, PowerPC, ALPHA et quelques autres architectures « haut de gamme », mais c'est encore hors de portée des x86-64 par exemple (qui ne disposent que de 15 registres utilisables).
- Ensuite, on pourrait enlever une opération de XOR en utilisant un polynôme avec un terme en moins.

Pour répondre à ces deux premières attentes, on pourrait réduire le nombre de termes du polynôme, c'est-à-dire utiliser un trinôme tel que *(15,1,0)* : nous économisons ici une opération et un mot de 64 bits. Cela n'a pas d'incidence importante sur l'efficacité de l'algorithme, puisqu'elle ne dépend pas du nombre de termes, et nous savons déjà (GLMF n°78 et 85) que le binôme de parité que nous venons d'enlever est inutile.

On peut aller plus loin en réduisant encore la taille du trinôme. Il est possible de choisir *(9,1,0)* (un autre trinôme primitif valide), qui a un cycle de 511 itérations. Comme la taille maximale des blocs est de 4096 octets, soit 512 mots de 64 bits, le registre ne devrait pas reboucler en cas d'erreur.

L'idéal serait d'utiliser un polynôme primitif dans `GF(2^8)`. Seulement voilà : il n'y a pas de polynôme

LISEZ-VOUS RÉGULIÈREMENT :

OFFRES DE COUPLAGE

Le magazine 100 % Linux

et / ou

Le magazine 100 % Sécurité

et / ou

100 % PRATIQUE

et / ou

Apprivoisez votre pingouin !

SI OUI, ALORS CES OFFRES D'ABONNEMENT À TARIF PRÉFÉRENTIEL VOUS SONT DESTINÉES...

11 N°s + 6 N°s — EN KIOSQUE(1) ~~100,80 €~~ **79 €**
soit une économie de 27,60 €

11 N°s + 6 N°s + 6 N°s — EN KIOSQUE(3) ~~154,80 €~~ **105 €**
soit une économie de 49,80 €

11 N°s + 6 N°s — EN KIOSQUE(2) ~~116,20 €~~ **83 €**
soit une économie de 33,20 €

11 N°s + 6 N°s + 6 N°s + 6 N°s — EN KIOSQUE(4) ~~190,30 €~~ **129 €**
soit une économie de 61,30 €

(1) Pour 11 N°s Linux Magazine + 6 N°s Linux Mag HS - (2) Pour 11 N°s Linux Magazine + 6 N°sMISC - (3) Pour 11 N°s Linux Magazine + 6 N°s MISC + 6 N°s Linux Mag. HS - (4) Pour 11 N°s Linux Magazine + 6 N°s MISC + 6 N°s Linux Mag. HS + 6 N°s Linux Pratique

OFFRE DE COUPLAGE À REMPLIR ET À RETOURNER À (OU PHOTOCOPIE)

LINUX MAGAZINE - BP 20142 - 67603 SELESTAT CEDEX

☐ **OUI, je m'abonne et désire profiter des offres spéciales de couplage**

Référence de l'offre :	Prix	Qté.	Total
☐ 11 N°s Linux Mag. + 6 N°s Linux Mag HS	79 €		
☐ 11 N°s Linux Mag. + 6 N°s MISC	83 €		
☐ 11 N°s Linux Mag. + 6 N°s MISC + 6 N°s Linux Mag HS	105 €		
☐ 11 N°s Linux Mag. + 6 N°s MISC + 6 N°s Linux Mag HS + 6 N°s Linux Pratique	129 €		
		TOTAL	

OFFRES VALABLES UNIQUEMENTS EN FRANCE MÉTRO.

Pour les tarifs étrangers, consultez notre site : www ed-diamond com

LES 4 FAÇONS DE VOUS ABONNER !

» PAR COURRIER POSTAL EN NOUS RENVOYANT LE BON CI-DESSOUS.

» PAR LE WEB, SUR NOTRE SITE : WWW.ED-DIAMOND.COM.

» PAR TÉLÉPHONE (PAIEMENT C.B.) ENTRE 9H-12H & 15H -18H AU 03 88 58 02 08.

» PAR FAX AU 03 88 58 02 09 C.B. ET/OU BON DE COMMANDE ADMINISTRATIF

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

2 Je joins mon réglement :

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

Boostez votre collection !

Avez-vous L'ÂME du COLLECTIONNEUR ?

VOUS RECHERCHEZ UN MAGAZINE EN PARTICULIER? ALLEZ SUR WWW.ED-DIAMOND.COM POUR VOIR LE SOMMAIRE DÉTAILLÉ DE CHAQUE MAGAZINE ET ENSUITE...

BOOSTEZ VOTRE COLLECTION AVEC LES "POWER PACKS X5", SOIT 5 LINUX MAGAZINE POUR 15€ ET LES "POWER PACKS X10", SOIT 10 LINUX MAGAZINE POUR 25€ À CHOISIR DANS LA LISTE CI-DESSOUS :

LES 4 FAÇONS DE COMMANDER !

- PAR COURRIER POSTAL EN NOUS RENVOYANT LE BON CI-DESSOUS.
- PAR LE WEB, SUR NOTRE SITE : WWW.ED-DIAMOND.COM.
- PAR TÉLÉPHONE (PAIEMENT C.B.) ENTRE 9H-12H & 15H -18H AU 03 88 58 02 08.
- PAR FAX AU 03 88 58 02 09 C.B. ET/OU BON DE COMMANDE ADMINISTRATIF

Choisissez vos numéros dans le tableau ci-dessous*

*SEULS LES NUMÉROS CI-DESSOUS SONT DISPONIBLES POUR UNE COMMANDE DE POWER PACKS PAR X5 ET X10

- N°06 GNOME - The Gimp
- N°07 Dopez Linux
- N°08 Le futur résolument objet
- N°09 Prêt pour le jeu !
- N°10 The HURD : 100% GNU
- N°11 Exclusif : l'avenir de G.N.O.M.E
- N°12 NT et Linux : Guerre ou complément ?
- N°13 Cryptage : la clé de la sécurité
- N°14 XFree 4.0 : le futur à notre portée
- N°15 Passez à la vitesse supérieure
- N°16 OpenSources : Est-ce suffisant?
- N°17 Linux : Système embarqué
- N°18 Spécial interview : l'avenir de Linux
- N°19 Dossier spécial : Postgre SQL 7.0
- N°21 Le protocole Internet du 21e siècle : IPv6
- N°22 Le multi-hreading : Une manière moderne de programmer le Multitâche
- N°23 Débugger sous Linux
- N°24 Palm et Linux
- N°25 Kernel 2.4.0
- N°26 <Dossier> XML </Dossier>
- N°27 Les systèmes de fichiers journalisés
- N°28 Scripting : la force d'Unix
- N°29 L.F.S. Linux From Scratch
- N°30 Le chiffrement des données
- N°31 VPN et tunneling
- N°32 Changez de coquille
- N°34 XSL - FO :TeX Killer ?
- N°35 QoS et iproute : optimisation et contrôle du trafic IP
- N°36 Linux embarqué : Le projet mGlinux
- N°37 L'impression sous Linux
- N°38 Le desktop Shell : Enlightenment
- N°39 Sécurité : Patchez votre noyau !
- N°40 MySQL : la base de donnée OpenSource
- N°41 Steganographieou l'art de la dissimulation de données
- N°42 Développez vos pilotes de périphérique
- N°43 Administrez facolement votre réseau SNMP
- N°44 Comprenez NetBios pour Maîtriser l'interopérabilité windows GNU Linux
- N°45 Cohabitation : UnDNS Bind dans un reseau Windows 2000
- N°46 Debian : Utilisez Samba avec le support ACL
- N°47 GNUstep : le petit frère de Mac OS X ?
- N°48 Caudium, votre prochain serveur Web !
- N°49 Après MySQL & PostgreSQL SAP DB : La base de données libre & puisante
- N°50 Créez un album Photo avec PHP ...et sans MySQL
- N°51 Boostez votre site Web avec XML grâce à XSLT, CSS & XPath
- N°52 Linux Temps réel où en est-on aujourd'hui ?
- N°53 Linux sur PDA : Linux dans votre poche !
- N°54 Maîtrisez LVM
- N°55 Intelligence Artificielle : Principes & programmation de jeux de stratégie classique
- N°56 Développez vos applications Mozilla avec XPFE & XPCOM
- N°57 Maîtrisez la gestion... Slots & Signaux ... des événements en C++
- N°58 Djbdns enfin une alternative viable à BIND !
- N°59 Zopix, Créez un CD "Live" Zope en 10 minutes !
- N°60 JBoss serveur d'applications J2EE OpenSource
- N°61 Découvrez MySQL 5 et les procédures stockées
- N°62 Créez votre OS, principe et implémentation
- N°63 Les threads : kernel 2.6 et 2.4
- N°64 Adamoto
- N°65 Théorie et pratique : Supervision avec Nagios
- N°66 Créez votre Distribution Live
- N°67 C# .NET
- N°68 Le crash disque vous guette
- N°69 La réponse de Sun à Linux ? SOLARIS 10
- N°70 Découvrez et comprenez la technologie GRID
- N°71 Présentation et installation du Hurd
- N°72 Services Web... C/C++ et gSOAP
- N°73 Compression théorie algorithmes et programmation
- N°74 VFS : Système de fichiers virtuel
- N°75 Tuning de code
- N°76 Algorithmes évolutionnistes
- N°77 Systèmes de fichiers chiffrés
- N°78 Bluetooth
- N°79 Sécurisation du Noyau avec PAX
- N°80 Run in memory
- N°81 Comment fonctionnent les générateurs de nombres pseudo-aléatoires
- N°82 eCos, une autre solution libre pour systèmes embarqués

NUMÉROS LINUX MAGAZINE ÉPUISÉS

N°01, N°02, N°03, N°04, N°05, N°20, N°33.

BON DE COMMANDE POWER PACKS À REMPLIR ET À RETOURNER À (OU PHOTOCOPIE)

LINUX MAGAZINE - BP 20142 - 67603 SELESTAT CEDEX

	OUI, je désire acquérir un POWER PACK X5	1er 1PP* X5	2ème 2PP* X5	3ème 3PP* X5
Cochez ici ▶ POWER PACKS X5	1, Linux Magazine N°			
	2, Linux Magazine N°			
	3, Linux Magazine N°			
	4, Linux Magazine N°			
	5, Linux Magazine N°			
	Total par série de POWER PACKS X5 :	15 €	30 €	45 €
	OUI, je désire acquérir un POWER PACK X10	1er 1PP* X10	2ème 2PP* X10	3ème 3PP* X10
Cochez ici ▶ POWER PACKS X10	1, Linux Magazine N°			
	2, Linux Magazine N°			
	3, Linux Magazine N°			
	4, Linux Magazine N°			
	5, Linux Magazine N°			
	6, Linux Magazine N°			
	7, Linux Magazine N°			
	8, Linux Magazine N°			
	9, Linux Magazine N°			
	10, Linux Magazine N°			
	Total par série de POWER PACKS X10 :	25 €	50 €	75 €
	Les Hors Séries et numéros spéciaux sont exclus des POWER PACKS. Montant TOTAL 15€ + 3,81€ de frais de port. Le TOTAL s'élève à 18,81€ pour l'achat d'un POWER Pack x5. SEULEMENT EN FRANCE MÉTROPOLITAINE ! *PP= POWER PACK	TOTAL :		
		Frais de port :		+3,81€
		TOTAL :		

1 Voici mes coordonnées postales

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

2 Je joins mon réglement :

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement par carte bancaire :

N° Carte :

Expire le : Cryptogramme Visuel : Voir image ci-dessous

Date et signature obligatoire : 200

primitif à 3 termes dans GF(2^8). Par contre, dans GF(2^7), il en existe deux ayant uniquement 3 termes : *(7,1,0)* et *(7,3,0)*, tous deux ayant les mêmes propriétés.

- Enfin, au point où nous en sommes, rien ne nous empêche d'utiliser des extensions 128 bits telles SSE, pour encore doubler la vitesse de calcul.

S'il n'est pas critique que le nombre d'itérations de la boucle n'est pas une puissance de deux, nous obtenons alors un code qui ressemblerait à celui-ci :

```
  U128 r0, r1, r2, r3, r4, r5, r6;
#define GFSR_ROUND128(a,b,p) \
 a = U128_XOR(p, U128_XOR(a, b));
/* GFSR de poly = (7,1,0) */
  GFSR_ROUND128(r6,r0,*p)
  GFSR_ROUND128(r5,r6,p[1])
  GFSR_ROUND128(r4,r5,p[2])
  GFSR_ROUND128(r3,r4,p[3])
  GFSR_ROUND128(r2,r3,p[4])
  GFSR_ROUND128(r1,r2,p[5])
  GFSR_ROUND128(r0,r1,p[6])
```

Ce code très simple réduit à seulement deux le nombre d'instructions SSE par itération, ce qu'il n'est plus possible d'améliorer. De plus, chaque instruction dépend directement de la précédente, ce qui bloque les unités d'exécution (elles passent leur temps à attendre le résultat des autres). On pourrait songer à entrelacer deux codes identiques, mais il n'y a plus assez de registres.

Et surtout, nous atteignons maintenant des limitations physiques, en particulier la bande passante entre le processeur et la mémoire : le code est tellement rapide qu'il est *memory bound* ! En fait, il est peu probable que les améliorations évoquées dans cette partie aient un effet notable sur la vitesse réelle du code, car le processeur ne peut probablement pas accéder aux données suffisamment vite.

Par exemple, considérons un processeur capable de traiter 8 octets par cycle à une fréquence de 2GHz : il faudrait alors 16G octets par seconde de bande passante avec la mémoire centrale. Les Front-Side Bus des processeurs les plus récents annoncent 3,2 à 6,4GO/s théoriques et certainement plus en interne (avec la mémoire cache), mais l'intérêt pratique de travailler uniquement en mémoire cache est réduit.

3. Construction de la fonction de signature avec ses compensations diverses

Il semble que même sans ces dernières optimisations extrêmes, notre code a atteint la limite pratique de performance (*point of diminishing return*). Mais d'autres écueils nous attendent, alors examinons maintenant les autres aspects de cet algorithme agressivement parallélisé. Car, pour pouvoir fonctionner à vitesse maximale, ce dernier a besoin de conditions externes qu'il va falloir réunir et assembler de manière méticuleuse et optimale.

3.1 Initialisation

Puisque la signature s'effectue sur des blocs indépendants (contrairement au générateur de nombres pseudo-aléatoires codé en mars 2006), nous n'avons pas besoin de conserver l'état des registres entre deux appels de la fonction, ce qui simplifie un peu le code. Mais cela est largement compensé par d'autres facteurs...

Si ce n'est toujours pas clair, répétons-le : il ne faut pas initialiser un registre de CRC à zéro, au risque de manquer des erreurs en début de bloc. Cela dit, toute valeur non nulle fait l'affaire, le standard 0xFFFF autant qu'autre chose. Toutefois, de nouveaux éléments entrent en compte :

- Nous utilisons un pseudo-CRC au lieu d'un CRC normal. Nous avons vu cet été (GLMF n°85) que la valeur initiale 0xFFFF a le même défaut que 0x0000 lorsque la configuration de Fibonacci est utilisée.
- Nous n'avons plus un, mais 64 registres parallèles.
- L'étape finale pourrait éventuellement provoquer des « collisions inattendues ». Nous allons étudier dans le prochain article comment « compacter » les 128 octets des registres en une signature de 2 octets seulement, mais cette opération pourrait induire des effets de bord malvenus.

Nous sommes dans une situation similaire à celle de l'initialisation du tGFSR de mars 2006, avec la contrainte supplémentaire de la fiabilité (ne pas créer des nouvelles formes de *faux positifs*) et de la vitesse : il faut utiliser le *minimum* d'opérations possible !

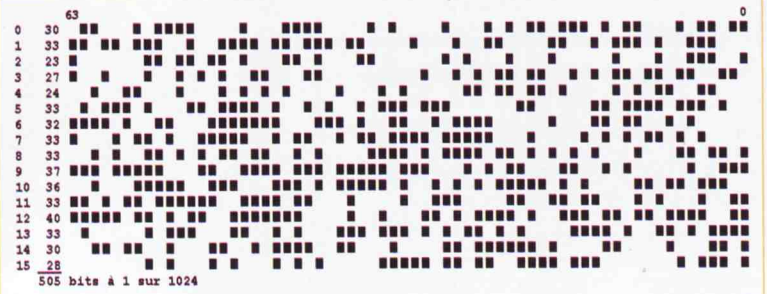
Le type de code utilisé pour le tGFSR semble convenir, mais il ne travaille que sur 32 bits et surtout il est trop lent : lors de la signature ou de la vérification d'un bloc, nous n'avons pas le temps de re-générer à chaque fois les mêmes données. Finalement, le plus simple est de préparer les valeurs initiales, puis de les recopier dans la zone de CRC pour chaque nouveau bloc. Elles peuvent donc être choisies avec plus de liberté et de soin, puis être codées directement comme des constantes.

Les valeurs suivantes sont obtenues à partir des signatures MD5 de différents fichiers, ce qui rend impossible toute corrélation entre elles.

```
U64 CRC16_64_const[16]={
  0x62f08f0a115bac5bULL, 0xdb93dbb089865d38ULL,
  0x81b68d1814441139ULL, 0x912a63004ad95b66ULL,
  0x2622a84506da1598ULL, 0x5d1bd29770306f74ULL,
  0xf4c7f1d32f046ca0ULL, 0x8b4f965bdf10ab50ULL,
  0x29ab650f5ed5446dULL, 0xef8cf77dc5632827ULL,
  0x23e71d7d2afa86dcULL, 0xd6fcf6213866c513ULL,
  0xfb59fc246bd3b6f3ULL, 0x4171947756a1e9afULL,
  0x36465e621bf4308dULL, 0x01451507db3dc05dULL
};
```

La **figure 3** montre visuellement la répartition des bits. Cela, ainsi que leur nombre biaisé (505 bits à 1, au lieu de 1024/2=512), est une première précaution contre d'éventuelles mauvaises surprises lors de la compaction, telles des recombinaisons entraînant des signatures problématiques dans les cas extrêmes.

Figure 3 : La constante servant à initialiser les registres de CRC16 ne devrait pas contenir de corrélations.



Cette table à très forte entropie n'a rien de magique. D'autres tables peuvent faire l'affaire, si elles satisfont (plus ou moins) les critères suivants : les valeurs des lignes de bits formées par les colonnes, les lignes et les diagonales doivent être différentes les unes des autres, et éviter les trop longues suites de 0 ou de 1 consécutifs. Cela rappelle un peu la conception des *S-tables* de l'algorithme de chiffrement *DES*...

3.2 Compensation de l'alignement

Si la boucle principale n'est pas affectée par l'alignement des données, et peut donc travailler à vitesse maximale, la difficulté a été reportée vers le prologue et l'épilogue.

Jusqu'à maintenant, nous avons étudié le mécanisme des CRC parallèles en considérant que le pointeur vers le bloc de données à signer est correctement aligné : si la granularité est de 8 octets, les 3 bits de poids faible de l'adresse sont à zéro. C'est une nécessité absolue, car les accès non alignés sont soit lents (ou très lents, selon les processeurs), soit interdits. Travailler avec des pointeurs non alignés est donc une impossibilité dans un code qui se veut portable.

Note de l'auteur : Un bébé manchot peut-il sucer son pouce ?

Habituellement, la première solution qui vient à l'esprit est d'effectuer la rotation soi-même, mot par mot dans la boucle de calcul principale. C'est une voie difficile pour plusieurs raisons, en particulier à cause de l'absence d'opérateur de rotation en C et dans les instructions MMX (une absurdité inacceptable). Cela rajoute aussi des instructions, qui doublent presque la taille de notre boucle principale, la ralentissant d'autant.

Heureusement, puisque les octets ne communiquent pas entre eux, il existe une autre solution : décaler l'origine, la référence. Un peu comme lorsque le paysage semble partir, alors que nous regardons par la fenêtre d'un train qui démarre... Cela revient à réassigner les octets dans les registres de CRC, pour qu'ils soient alignés avec les données à signer.

Figure 4 : L'ordre des octets est changé à l'intérieur des registres de CRC pour s'aligner sur la mémoire.

a) Un bloc de 22 octets

7	6	5	4	3	2	1	0
F	E	D	C	B	A	9	8
		15	14	13	12	11	10

b) Si le pointeur n'est pas aligné, par exemple ici avec un excès de 3 octets, les octets semblent décalés.

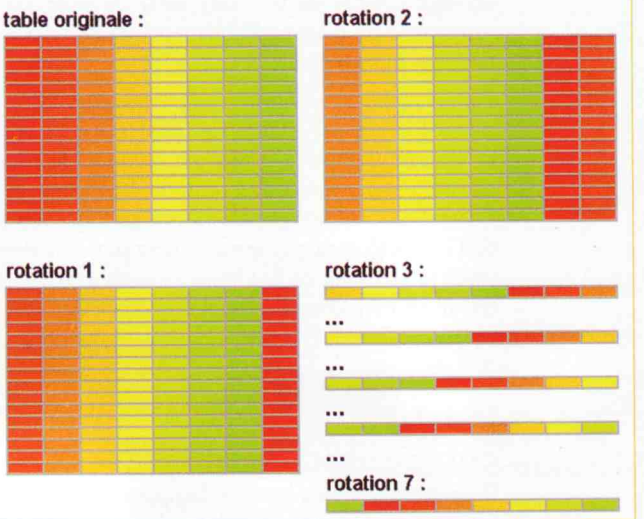
1 accès = 8 octets							
7	6	5	4	3	2	1	0
4	3	2	1	0			
C	B	A	9	8	7	6	5
14	13	12	11	10	F	E	D
							15

c) Dans les registres de CRC, l'ordre des octets a subi une rotation pour s'aligner sur celui de la mémoire.

7	6	5	4	3	2	1	0
4	3	2	1	0	7	6	5

La conséquence directe est que la table d'initialisation constante doit aussi subir une rotation. Et puisqu'il est trop long de calculer cette rotation à chaque fois, il faut précalculer 8 tables, une pour chaque rotation possible. De 128 octets, la taille totale des tables de constantes passe à 1024 octets. Cela reste raisonnable, puisqu'elles sont accédées sporadiquement et séquentiellement, alors qu'un CRC classique effectue un accès aléatoire par octet signé.

Figure 5 : Pour réduire au strict minimum les calculs au début d'une signature, on précalcule les 8 rotations de la table constante.



3.3 Compensation de l'Endian du processeur

L'équivalent informatique du fameux « *Et si on danse ?* » de Gaston Lagaffe, c'est l'Endian : l'ordre dans lequel les octets sont lus en mémoire et écrits dans un registre du processeur (et vice versa). Comme Gaston, on se

démène pour faire un algorithme impressionnant (ou un chouette costume pour le bal), mais on en oublie les interactions avec les autres (que ce soit avec d'autres processeurs ou des demoiselles). Ah, les geeks !

Figure 6 : À cause des deux organisations d'octets possibles, les informaticiens sont souvent schizophrènes (une maladie professionnelle encore ignorée).

En informatique, on aurait pu choisir une convention et vivre avec. Octets de poids faible aux adresses supérieures ou inférieures, il aurait suffi de se décider. Mais comme les deux sont possibles (sans parler des autres permutations !), les deux sont utilisés et la guerre de tranchées a éclaté entre pratiquants opposés. Le pire est qu'aucune vision n'a d'avantage décisif sur l'autre et tout le monde a donc raison, ce qui est un paradoxe avec lequel des générations de programmeurs ont dû apprendre à vivre, avec plus ou moins de bonheur.

À la longue, la domination écrasante du PC et de son petit Endian a plus ou moins éteint les flammes des émeutes (en imposant un couvre-feu ?), mais, il reste quand même beaucoup de dissidence. Dans l'embarqué, les architectures PowerPC et 68K sont bien implantées, sans parler des *architectures-caméléons* qui peuvent être configurées pour l'un ou pour l'autre (comme MIPS). Le Big Endian tient aussi son fief dans les réseaux, où le bit de poids fort est traditionnellement envoyé en premier (quoique...). La tendance est pourtant à la cohabitation, puisque chaque pas en direction d'un camp éloigne de l'autre, et les deux ont leurs avantages et sont bien implantés.

Pour simplifier les choses pour tout le monde, nous considérons que nos flux de données sont composés d'octets indépendants. Cela élimine déjà toute considération sur leur ordre dans les mots, le premier octet à être lu étant l'octet n°0, le suivant étant l'octet n°1... Pas de mot, pas d'Endian. C'est particulièrement pratique pour notre algorithme de CRC, puisque celui-ci ne traite justement que des blocs d'octets.

Pourtant, les octets sont traités par mots pour aller plus vite, et l'ordre reste donc important. En ce qui concerne la table des constantes, le compilateur va mettre les octets dans le bon ordre pour que le bit 0 du mot vienne s'écrire dans l'octet 0 du registre. Mais l'Endian entre en jeu lors de la lecture en mémoire des octets à signer. Pour refléter cela, l'ordre des octets dans la table des constantes d'initialisation doit être compatible, donc inversé lorsque le processeur est Big Endian (le lecteur attentif aura remarqué mon parti pris pour l'organisation Little Endian, malgré mon allégeance pour Motorola dans ma jeunesse).

Figure 7 : Selon l'Endian du processeur, la table de constantes doit aussi être modifiée pour refléter l'ordre des octets dans les registres.

Little Endian :							
7	6	5	4	3	2	1	0
6	5	4	3	2	1	0	7
5	4	3	2	1	0	7	6
4	3	2	1	0	7	6	5
3	2	1	0	7	6	5	4
2	1	0	7	6	5	4	3
1	0	7	6	5	4	3	2
0	7	6	5	4	3	2	1

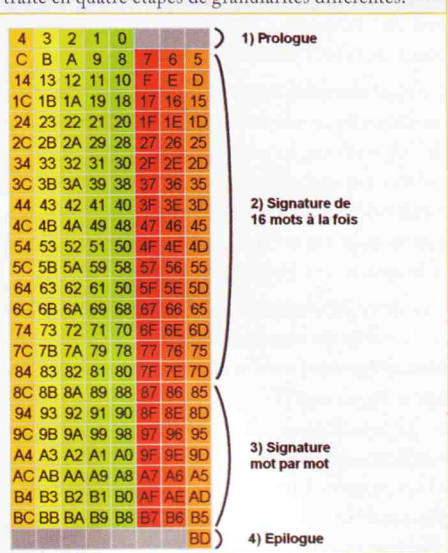
Big Endian :							
0	1	2	3	4	5	6	7
7	0	1	2	3	4	5	6
6	7	0	1	2	3	4	5
5	6	7	0	1	2	3	4
4	5	6	7	0	1	2	3
3	4	5	6	7	0	1	2
2	3	4	5	6	7	0	1
1	2	3	4	5	6	7	0

Une seule table est nécessaire à la fois, car on peut recalculer les valeurs lors de la compilation ou au démarrage du programme.

3.4 Construction de la routine de signature

À partir des informations précédentes, on peut commencer à construire la routine. Si le code semble complexe, c'est pour pouvoir traiter des blocs de toutes tailles, quel que soit l'alignement. Comme l'illustre la **figure 8**, le bloc est traité en 4 parties dont la taille et les propriétés varient.

Figure 8 : Un bloc de données, ici de 190 octets non alignés, est traité en quatre étapes de granularités différentes.



(1) Le premier mot doit pouvoir être calculé partiellement pour compenser l'alignement. C'est à cette étape que la bonne table de constantes est choisie. Cette partie doit aussi correctement gérer le cas particulier où la taille du bloc est inférieure à la taille du mot, sans recouvrir le suivant (comme illustré par la **figure 9**).

Figure 9 : Un cas particulier de bloc très court

(2) Ensuite, la routine va utiliser l'algorithme ultra-rapide déjà présenté dans la **partie 2.5**, et l'appliquer sur des blocs de 16 mots correctement alignés. Idéalement, il faudrait que cette partie soit exécutée au moins plusieurs fois, pour compenser la lenteur du prologue et de l'épilogue. Par rapport à la version initiale, cette partie subit quelques modifications subtiles, comme l'incrémentation préalable du pointeur pour compenser le premier calcul déjà effectué lors du prologue.

(3) S'il reste encore des mots alignés, ils sont traités (un par un) par l'algorithme itératif simple de `aligned_CRC16_64()` (**partie 2.2**), plus lent.

(4) Enfin, l'éventuel mot restant est traité et le résultat est masqué pour ne garder que les octets qui nous intéressent. C'est l'épilogue, déjà développé dans la **partie 2.4**.

Le texte de la fonction complète est si imposant qu'il a été déplacé dans une annexe à la fin de cet article. Ne vous forcez pas à lire le code linéairement, mais cherchez plutôt à reconnaître les morceaux déjà développés. La partie la plus délicate est le prologue, qui n'a pas encore été développé ou préparé. Il utilise cependant des techniques semblables à celles de l'épilogue.

Puisque les aspects théoriques sont déjà expliqués, la première difficulté réside surtout dans l'assemblage et la chasse aux bugs. Avec un code d'une telle sophistication, il était inévitable de trouver *quelques bulles d'air dans le ciment* ! J'ai validé ce code avec un peu de méthode, de nombreuses séances de `gdb` et plein de tests différentiels.

Ce n'est qu'ensuite que j'ai commencé à réorganiser le prologue pour rogner un peu de temps au démarrage de l'algorithme. Les instructions ont été entrelacées (en préservant leur ordre) pour maximiser le parallélisme d'exécution, au risque de rendre le tout... un peu confus. Les nombreux commentaires achèvent d'augmenter la longueur du code source.

Une des optimisations les plus étonnantes est la recopie de la table de constantes, qui a d'abord été réalisée manuellement, mot par mot. Sur x86, `GCC` convertit une assignation d'un mot de 64 bits en 4 opérations de 32 bits, 2 pour lire et 2 pour écrire, ce qui est assez inefficace quand on dispose des instructions MMX. Mais curieusement, c'est l'extension SSE qui a été la plus facile à utiliser, puisque `GCC` met à notre disposition plusieurs intrinsèques de déplacement, très pratiques ici :

```
#if defined (__SSE__)  /* version 128 bits */
#define MOVTABLE(index){                                    \
__builtin_ia32_storeups((float*)(CRC16_64_table+index) , \
 __builtin_ia32_loadups((float*)     (ptr_table+index)));}
```

La seule subtilité syntaxique consiste à *caster* les valeurs pointées par `ptr_table` vers des nombres de type *float*, puisque c'est ce qui est attendu (sans pourtant s'en préoccuper) par les intrinsèques. Avec cette technique, les entrées de la table originale sont copiées deux par deux avec seulement deux instructions. On raccourcit ainsi le prologue de 16 instructions !

Pour la version MMX (tout le monde n'a pas de Pentium III), un problème épineux se pose : il n'y a pas d'intrinsèques correspondant aux instructions `movq`. On pourrait utiliser l'assembleur (comme dans la définition de `U64_MOVD`), mais je trouvais ça trop lourd.

J'ai trouvé une solution plus élégante en traînant dans des archives de code source. Par endroit, j'ai vu que les variables pouvaient être associées à des registres spécifiques au moyen d'une syntaxe peu courante. Les exemples que j'ai trouvés utilisaient des registres normaux. J'ai donc tenté avec un nom de registre MMX et... bingo !

```
#if defined (__MMX__) /* version 64 bits forcée */
  #define MOVTABLE(index) {      \
    register U64 y asm("mm0");   \
    register U64 z asm("mm1");   \
    y = ptr_table[index];        \
    z = ptr_table[index+1];      \
    CRC16_64_table[index]   = y; \
    CRC16_64_table[index+1] = z; }
```

Les instructions de mouvement `movq` sont générées automatiquement par `gcc` qui finalement donne le résultat tant attendu.

Cette technique sera utilisée dans d'autres fonctions plus tard, lorsque `gcc` se remet à passer les données par la pile ou par les registres. Il faut toutefois faire attention à ne pas en abuser ! Le nombre de registres ainsi réservés doit rester limité pour laisser au compilateur assez de registres de travail (sinon, il se remettra à utiliser la pile). Et dans le cas présent, le nombre de registres disponibles est juste suffisant pour que ça marche.

3.5 Correction du déphasage et génération de la table

Arrivé à ce stade, on se sent fier d'avoir aplani (ou même compris) tant de difficultés, mais beaucoup d'autres astuces sont encore nécessaires ! La routine cache une incohérence temporelle, un cas subtil que nous devons encore traiter au moyen d'un précalcul plus complexe que prévu. Une autre contrainte est que la même table constante doit être utilisée par toutes les versions de l'algorithme, afin de simplifier le développement et les vérifications croisées.

Pour comprendre le problème que pose la routine `unaligned_CRC16_64()`, revenons à son prologue, qui traite sélectivement certains octets.

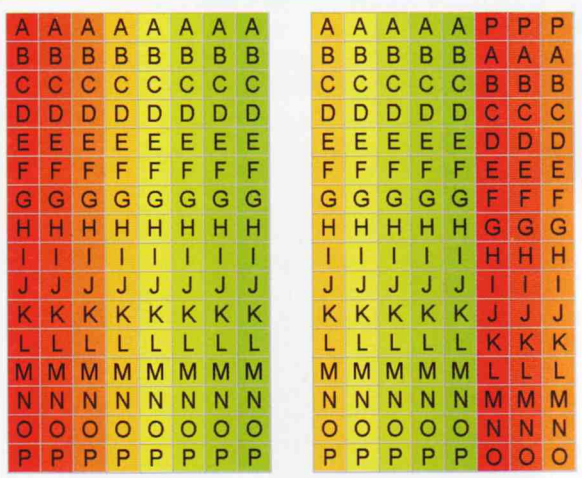
```
T0:
/* Aligne q sur 8 octets : */
  q = (U8*) ((PTR_CAST q) & -8L);
/* Création du masque : */
  mask = LOCAL_SHLr(mask, (offset<<3));
/* Lecture des 8 premiers octets (alignés) */
  t = *(U64*)q;
  q+=8;
.......
  r0 = ptr_table[0];  CRC16_64_table[0] = r0;
.......

T1:
/* Premier calcul de CRC */
  t  = U64_XOR( U64_XOR(r0,t), U64_XOR(r3,r2));
.......
/* réécrit juste les octets spécifiés par le masque */
  t    = U64_AND(t,mask);
  mask = U64_ANDN(mask, r0);
  r0   = U64_OR(mask, t);
  CRC16_64_table[0] = r0;
```

Si on regarde la valeur finale de `r0`, on s'aperçoit qu'elle contient des valeurs de T0 et de T1 mélangées, c'est-à-dire une combinaison du `r0` initial et du `t` calculé (lui-même dépendant de `r0` et de la donnée à signer). Au calcul suivant, la valeur devient encore plus fausse pour la partie masquée au début. Le calcul est effectué avec les mauvaises valeurs, alors que la partie non masquée est correcte.

Si on regarde orthogonalement, en considérant que chaque octet de la table est un bit d'un registre à décalage vertical, on s'aperçoit que tous les registres sont décalés (car l'index global est implicitement mis à jour), mais la *valeur* est mise à jour sur certains registres seulement.

Pour corriger cela, il suffit de *pré-décaler* les registres dont la valeur ne sera pas mise à jour lors du premier cycle. Heureusement, la correction a des conséquences minimes sur le code de précalcul, puisqu'il suffit d'ajouter une ligne en C.

Figure 10 : Lorsque le bloc à signer n'est pas aligné, il faut aussi corriger l'index de certains registres à décalage. Attention ! La correction dépend aussi de l'Endian et du sens de balayage des registres de CRC.



a) table originale

b) table après rotation *et* déphasage

```
/* Constantes arbitraire pour l'initialisation
 des registres de CRC, + 7 rotations de la table
 à 16 entrées en 64 bits, soit 1024 octets.
 Seules les premières entrées sont initialisées : */
U64 CRC16_64_const[16*8]={
#ifdef TESTS_TABLE
/* table spéciale pour vérifier l'algorithme */
  0x0102030405060708ULL, 0x1112131415161718ULL,
  0x2122232425262728ULL, 0x3132333435363738ULL,
  0x4142434445464748ULL, 0x5152535455565758ULL,
  0x6162636465666768ULL, 0x7172737475767778ULL,
  0x8182838485868788ULL, 0x9192939495969798ULL,
  0xA1A2A3A4A5A6A7A8ULL, 0xB1B2B3B4B5B6B7B8ULL,
  0xC1C2C3C4C5C6C7C8ULL, 0xD1D2D3D4D5D6D7D8ULL,
  0xE1E2E3E4E5E6E7E8ULL, 0xF1F2F3F4F5F6F7F8ULL
#else
/* table normale décorrélée */
  0x62f08f0a115bac5bULL, 0xdb93dbb089865d38ULL,
  0x81b68d1814441139ULL, 0x912a63004ad95b66ULL,
  0x2622a84506da1598ULL, 0x5d1bd29770306f74ULL,
  0xf4c7f1d32f046ca0ULL, 0x8b4f965bdf10ab50ULL,
  0x29ab650f5ed5446dULL, 0xef8cf77dc5632827ULL,
  0x23e71d7d2afa86dcULL, 0xd6fcf6213866c513ULL,
  0xfb59fc246bd3b6f3ULL, 0x4171947756a1e9afULL,
  0x36465e621bf4308dULL, 0x01451507db3dc05dULL
#endif
/* la suite est complétée par rotation_table() */
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
};
/* Précalcul des rotations de la table */
void rotation_table(void) {
  #if defined (__MMX__) /* version 64 bits forcée */
      register U64 t asm("mm0");
      register U64 u asm("mm1");
      register U64 v asm("mm2");
  #else
    U64 t, u, v;
  #endif
  int i=8, i2=56,
  /* à la fois compteurs et opérandes de décalage */

      j, /* compteur et index dans la table source */
      k=16; /* index dans la table de destination */
  /* recopie chaque entrée et effectue
   la rotation dans le bon sens */
  do {
    j=0; do {
      t=CRC16_64_const[j];        /* original */
      u=CRC16_64_const[(j-1)&15]; /* décalé d'un cran */
      j++;
      /* attention : pas de correction du shift */
      v=U64_OR(U64_SHLr(t,i), U64_SHRr(u,i2));
      CRC16_64_const[k]= CONVERT_LE64(v);
      k++;
    } while (j<16);
    i+=8, i2-=8;
  } while (i2!=0);
  /* Entrée initiale = pas de rotation à faire
   SAUF si Big Endian (mais alors en dernier car
   l'opération est faite "sur place") */
  #if __BYTE_ORDER == __BIG_ENDIAN
  for (k=0; k<16; k++)
    CRC16_64_const[k] = CONVERT_LE64(CRC16_64_
const[k]);
  #endif
}
```

La table des constantes a subi une petite modification supplémentaire : sa taille a été augmentée et on compte sur la routine d'initialisation pour remplir les zones vides, corriger l'Endian et effectuer le prédécalage. La modification est effectuée *sur place* pour économiser un peu d'espace. Cela rend `rotation_table()` un peu plus complexe, mais le prologue de la fonction de signature est simplifié. Le piège stupide à éviter est bien sûr l'oubli de l'appel à `rotation_table()` au début du programme.

3.6 Vérifications et portabilité

Le code a été initialement conçu pour `gcc3.3` sur Pentium III (Little Endian 32bits+MMX+SSE), puis porté sur Alpha/EV6 (Little Endian 64 bits natif) et SUN/UltraSparcIIi (Big Endian 64 bits). Ce port sur Big Endian n'a révélé que quelques erreurs minimes, mais elles étaient très subtiles. Quand on change l'ordre des octets, il faut se redemander quels mots il faut décaler et dans quels sens ou bien s'il faut carrément changer l'Endian du mot... Développer (sans préparation) un morceau de code aussi compliqué que celui-ci peut rendre schizophrène, ce qui justifie les explications détaillées des articles de cette série.

L'archive du code source contient un test exhaustif, que l'on peut exécuter pour vérifier qu'un portage s'est correctement déroulé. Plus d'une dizaine de personnes ont eu la gentillesse de le faire tourner sur leurs machines, ce qui a beaucoup augmenté la qualité du code et fourni des informations précieuses.

Tout d'abord, le code source est tout à fait valide sous `gcc2.95` tant que les extensions MMX/SSE sont désactivées. De nombreux systèmes embarqués dépendent encore de cette vieille version du compilateur, ils pourront donc utiliser ce code avec une petite pénalité de performance (absence des intrinsèques MMX/SSE et de l'option `-Os`).

Ensuite, pour porter un programme sur un maximum de machines, il est conseillé d'utiliser `autoconf/autotools`, mais j'ai jugé cela trop lourd pour un si petit programme. J'ai donc créé initialement un petit script de détection des paramètres essentiels, qui a rapidement évolué et grossi au fur et à mesure des tests sur des plateformes diverses : GNU/Linux, xBSD, HP-UX, OSX/Darwin, AIX, IRIX...

Je me suis d'ailleurs rendu compte après coup que je n'avais pas testé l'algorithme sur la plate-forme non-Unix la plus connue, celle conçue à Redmond. Je ne pense pas que ce portage pose de problème particulier, puisque les outils GNU y ont été portés (avec `cygwin` par exemple). Les intrinsèques x86 de `gcc` utilisent le standard Intel et devraient donc être compatibles avec les compilateurs propriétaires (Intel ou Microsoft).

Quelques compilateurs autres que `gcc` ont été testés avec un succès relatif. Il y a quelques différences superficielles de syntaxe, comme le support des fonctions *inline* ou bien le forçage de la taille des valeurs constantes (`0x... ULL`) par exemple. L'algorithme lui-même n'a pas été remis en question.

Mais l'ensemble des éléments de l'ordinateur influe sur le programme, pas seulement le processeur ou le compilateur, mais aussi le système de fichiers, le *shell*, les bibliothèques partagées, la révision de la machine... Le script de test dépend de tout cela pour activer les bonnes options de compilation, donner la bonne taille aux variables et détecter l'ordre des octets (quand cela est possible). L'intervention humaine reste donc nécessaire, surtout sur les plateformes non-GNU/Linux.

Évidemment, `autoconf` aiderait beaucoup. Toutefois, cet outil ne permet pas de tester l'algorithme lui-même et sa portabilité sur des architectures variées. En revanche, j'ai adopté quelques conventions qui faciliteront l'intégration de l'algorithme dans un programme utilisant `configure`. En particulier, j'ai remplacé

```
#if __BYTE_ORDER == __BIG_ENDIAN
```

par

```
#ifdef WORDS_BIGENDIAN
```

L'autre épineux problème (qu'aucun outil, à l'exception d'une conception soignée, ne peut résoudre) est la taille des entiers. La complexité de la situation est illustrée par le **tableau 1**, qui est loin d'être exhaustif.

Table 1 : Exemple de configurations matérielles et logicielles

Machine	CPU	Système	Taille Des registres	Endian	sizeof()				
					char	short int	int	long int	long long int
Intel	x86	MS-DOS/BorlandC	16 bits	LE	1	2	2	4	4
Intel	x86 / ia32	GNU/Linux	32 bits	LE	1	2	4	4	8
SGI (IP22)	MIPS R4400	irix6.2/gcc2.95.2	32 bits	BE	1	2	4	4	8
HP	PA-RISC	HP-UX11	64 bits	BE	1	2	4	4	8
Mac	Power G5	Darwin/gcc4.0.1	64 bits	BE	1	2	4	4	8
SUN	UltraSparcIIi	GNU/Linux	64 bits	BE	1	2	4	4	8
Intel	Itanium 2	GNU/Linux	64 bits	LE	1	2	4	8	8
AMD	x86-64	GNU/Linux	64 bits	LE	1	2	4	8	8
Compaq	Alpha	GNU/Linux	64 bits	LE	1	2	4	8	8
Cray	YMP-EL	UNICOS9.0/CrayC	64 bits	BE	1	8	8	8	8

En bref, contrairement à la plupart des autres langages, il n'y a pas de taille bien définie en C... C'est même carrément n'importe quoi ! Dans le code source, nous utilisons les définitions **U8**, **U16**, **U32**, etc. pour les variables, mais on ne peut pas s'arrêter là : d'une part, il faut assigner le bon nom de taille, d'autre part, la bonne taille n'existe peut-être pas !

En fait, la plupart des tests de portabilité ont initialement échoué à cause d'une incompatibilité de taille des registres. Finalement, j'ai mis au point un petit système où `def64.h` inclut le fichier `machine.h`, lui-même généré par le programme contenu dans `machine.c` :

```
/*
   fichier machine.c
   version 22 janvier 2007 par whygee@f-cpu.org
Ce petit programme est un remplaçant d'autoconf
qui détecte les paramètres critiques du système
sur lequel il tourne. Il génère le fichier machine.h
qui est utilisé par def64.h, mais il peut être
édité pour correspondre à d'autres architectures cibles.
Note : les variables sont prévues pour supporter
"au moins" le nombre de bits voulu, le code de CRC16
semble s'en accomoder.
 */

#include <stdio.h>

/* La chaine de caractères "87654321"/"12345678" */
unsigned long long int i[2]={0x3837363534333231,
  0}; /* pour le zero terminal */

int main(int argc, char *argv[]) {
  char *c=(char *)&i,
    *endian="#error UNDEFINED ENDIAN",
    *u32="#error UNDEFINED U32";
  int j;
  int ssi, si, sli;

  ssi = sizeof(short int);
   si = sizeof(int);
  sli = sizeof(long int);

  if (sizeof(char) !=1 ) {
    printf("#error sizeof(char)=%i!=1\n",
       (int)sizeof(char));
  }

  if (ssi < 2) {
    printf("#error sizeof(short int)=%i<2\n",
       (int)ssi);
  }

  if (si >= 4) {
    u32="int";
  }
  else {
    if (sli >= 4) {
      u32="long int";
    }
    else {
      printf("#error incapable de trouver \
le nom d'un type 32 bits\n");
      printf("#error sizeof(int)=%i, \
sizeof(long int)=%i\n", (int)si, (int)sli);
    }
  }
  if (sizeof(long long int)!=8) {
    printf("#error sizeof(long long int)=%i!=8\n",
       (int)sizeof(long long int));
  }
  if ((c[0]=='8') && (c[1]=='7') && (c[2]=='6')
   && (c[3]=='5') && (c[4]=='4') && (c[5]=='3')
   && (c[6]=='2') && (c[7]=='1') && (c[8]==0)) {
    endian="#define WORDS_BIGENDIAN";
  }
  else {
    if ((c[0]=='1') && (c[1]=='2') && (c[2]=='3')
     && (c[3]=='4') && (c[4]=='5') && (c[5]=='6')
     && (c[6]=='7') && (c[7]=='8') && (c[8]==0)) {
      endian="/* Little Endian detected, \
#undef WORDS_BIGENDIAN */";
    }
    else {
      printf("#error Endian=%s ???\n",c);
    }
  }

  printf("\
/* fichier machine.h\n\
 Attention ! Ce fichier a été généré par machine.c");

/* inclusion d'infos complémentaires */
  if (argc>1) {
    printf("\n tag : ");
    for (j=1; j<argc; j++)
      printf("%s ",argv[j]);
        /* faille de sécurité en perspective */
  }

  printf("\n*/\n\
\n\
#ifndef __MACHINE_H__\n\
#define __MACHINE_H__\n\
\n\
/* sizeof(short int)=%d\n\
   sizeof(      int)=%d\n\
   sizeof(long  int)=%d */\n\
", ssi, si, sli);

  printf("\n\
#define U8    unsigned char\n\
#define S8      signed char\n\
\n\
#define U16   unsigned short int\n\
#define S16     signed short int\n\
\n\
#define U32   unsigned %s\n\
#define S32     signed %s\n\
\n\
#define U64   unsigned long long int\n\
#define S64     signed long long int\n\
\n\
/* Endian : */\n\
#define ENDIAN64 %s\n\
%s\n\
\n\
#endif\n\
", u32, u32, c, endian);

  return 0;
}
```

Sur tous les essais que j'ai effectués, les tailles sur lesquelles j'ai pu me reposer sont `char` et `long long int`. Les autres variables de `unaligned_CRC16_64()` sont soit des compteurs de boucle (qui ne dépassent pas 64K), soit des pointeurs (qui sont donc automatiquement à la taille des registres de la machine).

Pour réduire la difficulté du portage, j'ai restreint les valeurs possibles au minimum et vérifié que le code fonctionne encore si on utilise un type de donnée dont la taille est supérieure à celle qu'on veut. Par exemple, imaginons que la plate-forme cible ne supporte pas le type **U16** et emploie **U32** à la place : cette taille est utilisée uniquement pour retourner le résultat de la fonction. Il suffit alors de masquer ses bits de poids fort pour retourner un résultat valide.

```
U16 crc16_64() {
  ....
  return resultat & 0xFFFF;
}
```

Évidemment, il faut faire attention à la granularité de l'accès à la mémoire, par exemple éviter qu'un accès **U32** lise 64 bits d'un coup. La méthode la plus simple est d'accéder à la mémoire octet par octet, ce qui est aussi indépendant de l'Endian du processeur. Mais c'est vrai qu'en disant qu'un entier **U32** fait « au moins 32 bits », on est encore moins sûr de quoi que ce soit, du coup !

Le test le plus édifiant s'est déroulé sur un Cray YMP-EL, grâce au libre service fourni par `cray-cyber.org`. L'architecture des ordinateurs vectoriels Cray est très, très spéciale, car elle ne supporte qu'une seule taille de mots, sur 64 bits. Les octets sont émulés par des décalages/masques et il n'y a qu'une seule taille pour les **short int**, **int**, **long int** et **long long int** : 8 ! (*Au fait, 8 quoi, d'ailleurs ?*)

Cela brise les hypothèses sur lesquelles est bâti le code, en particulier la granularité d'adressage sur octets. Ce dernier point modifie complètement l'arithmétique des pointeurs et les modifications nécessaires au code source ont été jugées trop lourdes pour justifier le portage immédiat. C'est d'ailleurs une plate-forme peu représentative des systèmes embarqués et trop peu répandue (mais tellement... délirante).

Cependant, certains processeurs embarqués très spéciaux empruntent cette caractéristique d'accès par mots, sans pointeurs sur octets : ce sont les **DSP**, tels ADSP21xx/21k, Motorola 56k ou TMS320Cxxx pour les plus connus. Par exemple, la série ADSP2106x (la famille *SHARC*) accède à des mots de 16, 32 ou 40 bits en fonction du bloc mémoire interne pointé et de sa configuration. L'ordre des octets sur le bus mémoire externe est aussi très *contre-intuitif*. Ces processeurs spécialisés sont des plateformes formidables pour le traitement du signal en temps réel. Nous nous y intéresserons donc plus tard dans la série d'articles.

3.7 De nouvelles perspectives d'optimisation

À force de modifications et de transformations, l'algorithme de calcul de signature (qu'on ne peut plus vraiment appeler CRC) finit par tenir dans les registres de la majorité des processeurs. Ainsi, le problème original qui était *memory-bound* est devenu *CPU-bound* si on estime que les données à signer peuvent être lues suffisamment vite (ce qui est encore discutable, mais admettons-le pour l'exemple).

Revenons sur la problématique de la **partie 2.6**, en considérant que la mémoire est assez rapide et que le processeur peut effectuer plusieurs opérations en parallèle. Nous avons vu que le calcul est freiné par les dépendances avec les résultats antérieurs :

```
#define CRC_ROUND_U8(x, a, b, c)                \
  r##b = CRC16_64_table[(x+1) & 15];            \
  r##a = U64_XOR( U64_XOR( r##a , ((U64*)q)[x-1]), \
                  U64_XOR( r##b , r##c ));      \
  CRC16_64_table[(x) & 15] = r##a ;
```

Le résultat de `r##a` dépend de deux autres résultats, dont un dépend d'une lecture. Comme expliqué dans GLMF n°85 de juillet 2006, nous sommes arrivés à cet algorithme, car il est trop lourd d'accéder en *lecture/modification/écriture* à plusieurs éléments en mémoire simultanément. Or l'algorithme de *pipeline* (**partie 2.5**) optimise les accès à la mémoire en gardant les données immédiatement utilisées dans les registres. Il n'est donc plus nécessaire d'utiliser la configuration de Fibonacci. En utilisant à nouveau la configuration de Galois, nous retournons dans la théorie initiale des CRC (plus familière).

Convertir le code en configuration de Galois est très simple et ne modifie pas la structure générale du source. En allouant un registre supplémentaire pour contenir la donnée à signer, il est maintenant possible d'effectuer les 3 opérations de XOR en parallèle et donc de débloquer les dépendances qui ralentissaient encore le processeur. Dans l'exemple suivant, nous réutilisons la variable `t` :

```
#define CRC_ROUND_GALOIS(x, a, b, c)      \
  r##b = CRC16_64_table[(x+1) & 15];      \ (1)
  t = ((U64*)q)[x-1];                     \ (2)
  r##a = U64_XOR( r##a , t);              \ (3)
  r##b = U64_XOR( r##b , t);              \ (4)
  r##c = U64_XOR( r##c , t);              \ (5)
  CRC16_64_table[(x) & 15] = r##a ;         (6)
```

Cela rajoute une instruction supplémentaire dans du code x86/MMX, mais l'opération de lecture est effectuée implicitement dans l'instruction composée originale, et séparément sur les autres architectures, il n'y a donc pas de pénalité. En fait, cette forme de code permet même d'autres types d'optimisation, par exemple en réordonnant les instructions pour les Pentium MMX. Une des nombreuses contraintes de ce processeur superscalaire statique est de ne pouvoir effectuer qu'un accès mémoire MMX par cycle (toutes les 2 instructions), et nous avons ici le mélange d'instructions idéal pour entrelacer les opérations (3 xor indépendants et 3 accès mémoire).

```
#define CRC_ROUND_GALOIS_PMMX(x, a, b, c) \
paire 1:
  t = ((U64*)q)[x-1];                     \ (2)
  slot vide car t est inutilisable ici
paire 2:
  r##b = CRC16_64_table[(x+1) & 15];      \ (1)
  r##a = U64_XOR( r##a , t);              \ (3)
```

```
paire 3:
  r##b = U64_XOR( r##b , t);              \ (4)
  r##c = U64_XOR( r##c , t);              \ (5)
paire 4:
  CRC16_64_table[(x) & 15] = r##a ;         (6)
  slot vide
```

En échangeant les instructions 1 et 2, le nouveau bloc de code s'exécute en 4 cycles au lieu de 5 sur Pentium MMX. Des modifications plus complexes permettraient de descendre à 3, au prix de macros illisibles, ainsi que d'un prologue et d'un épilogue encombrants. Beaucoup d'efforts, simplement à cause d'une règle stupide de pairage des instructions sur un processeur obsolète...

Voyons si on peut repartir d'un code plus simple.

Le code à 7 registres décrit dans la **partie 2.6** est très difficile à faire descendre en dessous de 2 cycles processeur par calcul, même si la lecture en mémoire était possible dans n'importe lequel des deux pipelines du Pentium MMX. Il faut en effet un cycle pour l'accès mémoire, puis un cycle pour le calcul des deux XOR.

```
  U64 r0, r1, r2, r3, r4, r5, r6;
#define GFSR_ROUND64_7(a,b,p)     \
paire 1:
 t = p; /* accès à la mémoire */  \
paire 2:
 a = U64_XOR(a, t);               \
 b = U64_XOR(b, t);               \
  GFSR_ROUND64_7(r6,r0,*p)
  GFSR_ROUND64_7(r5,r6,p[1])
  GFSR_ROUND64_7(r4,r5,p[2])
  GFSR_ROUND64_7(r3,r4,p[3])
  GFSR_ROUND64_7(r2,r3,p[4])
  GFSR_ROUND64_7(r1,r2,p[5])
  GFSR_ROUND64_7(r0,r1,p[6])
```

Pour les processeurs OOO, on peut encore améliorer la performance par quelques détails. Par exemple, le polynôme choisi *(7,1,0)* implique la réutilisation immédiate du résultat d'un calcul précédent. Dans le code ci-dessus, la variable `b` devient `a` deux instructions plus tard, mais le résultat n'aura peut-être pas eu le temps d'être calculé. Le polynôme *(7,3,0)* laisserait un peu plus d'espace/temps entre les calculs et réduit le risque de dépendances bloquantes.

```
  GFSR_ROUND64_7(r6,r2,*p)
  GFSR_ROUND64_7(r5,r1,p[1])
  GFSR_ROUND64_7(r4,r0,p[2])
  GFSR_ROUND64_7(r3,r6,p[3])
  GFSR_ROUND64_7(r2,r5,p[4])
  GFSR_ROUND64_7(r1,r4,p[5])
  GFSR_ROUND64_7(r0,r3,p[6])
```

À ce niveau, les optimisations dépendent de plus en plus de la plate-forme et de ses particularités. Un code optimal pour Pentium MMX ne sera pas optimal sur un Pentium II/Pentium III et encore moins sur Pentium IV. Par contre, le code ci-dessus, bien qu'il ait peu de chances de tourner à moins de deux cycles par itération sur Pentium MMX, pourrait atteindre 1,5 ou 1 cycle par itération sur d'autres processeurs plus récents. On n'en finit jamais !

4. Constatations

Pour des raisons rédactionnelles évidentes, j'abrège cet article avec la frustration de laisser en suspens la finalisation du programme, car il n'est pas encore complet. L'étape de compaction est indispensable pour réduire la taille de la signature, de 128 à 2 octets, sans perdre ni de temps, ni d'informations importantes. Nous examinerons les algorithmes nécessaires dans le prochain opus de cette saga. Mais nous avons déjà abordé un certain nombre de techniques et de points très intéressants :

- La combinaison des techniques de compensation de l'Endian et de l'alignement permet de réaliser du code *à la fois* rapide et portable. C'est presque l'équivalent informatique de la démonstration de la quadrature du cercle. Leur maîtrise est indispensable et leur réutilisation est prévue pour d'autres types d'algorithmes.
- `cpp` est outrageusement limité, vive `m4` !
- Les fonctions intrinsèques de GCC (`__builtin_ia32_XXXX()`) permettent d'améliorer la performance du code, dans le cadre d'une démarche d'optimisation globale et approfondie. Elles évitent l'emploi des directives `__asm__();`, ainsi que le casse-tête de l'allocation des registres, et favorisent la portabilité. Mais il ne faut pas oublier de **constamment vérifier le travail du compilateur** (en appelant `gcc -S`), pour s'assurer que le résultat est bien celui attendu.
- `gcc -Os` devrait être l'option par défaut du compilateur. Le résultat n'est pas toujours parfait, mais est souvent bien meilleur que les autres méthodes d'optimisation, qui ont la fâcheuse manie de passer par la mémoire pour des broutilles.
- Réaliser un programme portable est très difficile. L'expérience ne suffit pas, il faut aussi constamment vérifier la validité des codes d'autotests sur de nombreuses machines différentes. La portabilité implique aussi de nombreux compromis et des solutions parfois sous-optimales. En revanche, un algorithme portable sera certainement plus utile et répandu qu'un algorithme trop spécifique à un type de machine. Ainsi, se concentrer sur l'optimisation d'un seul morceau de code est plus rentable que de créer de nombreuses versions (comme dans l'article sur le CRC16 de GLMF n°78). Enfin, les compromis nécessaires à la portabilité incitent parfois à chercher des solutions plus simples et plus générales, ce qui évite paradoxalement la complication galopante du code source.

Conclusion

Cet article a réutilisé des notions développées depuis décembre 2005 (GLMF n°78) pour construire un algorithme très sophistiqué, en essayant de garder une réalisation simple, efficace et flexible. Bien sûr, ce n'est pas parfait, mais les techniques présentées ici sont plus avancées que celles présentées dans la littérature traditionnelle.

Par rapport à la collection de routines publiée en décembre, notre nouveau super-CRC a des avantages importants, qu'il faut toutefois tempérer :

- Encombrement mémoire : de 1024 ou 512, on n'utilise plus que 128 octets de mémoire temporaire. La table prédécalée utilise aussi 1024 octets, mais elle n'est pas accédée constamment ou dans un ordre aléatoire, elle peut donc résider en cache L2.
- L'étape de précalcul (bit à bit) de la LUT du CRC16 a été remplacée par les prédécalages.
- L'algorithme est indépendant de l'Endian du processeur, la vitesse de traitement ne varie pas dramatiquement selon le type de ce dernier. L'ordre des octets est corrigé en aval du calcul et il n'y a qu'une seule version de l'algorithme principal à développer. Les variations d'Endian sont presque toutes masquées par des macros spécifiques placées dans un autre fichier.
- Le choix d'une largeur de 64 bits permet aux processeurs récents de fonctionner à plein rendement. Les processeurs 32 ou 16 bits ne devraient pas être trop pénalisés non plus, bien que le calcul d'une signature nécessite deux ou quatre fois plus d'instructions. Des versions 16 et 32 bits sont réalisables de plusieurs manières, selon la taille du banc de registres. Nous verrons ces techniques dans d'autres articles à venir.
- Le surcoût de la compaction après le calcul est loin d'être négligeable et réserve ce type d'algorithme à des blocs d'au moins 1K octets.
- Cette méthode n'est compatible avec aucune autre, alors que celle présentée en décembre 2005 calcule un CRC standard.

Mais on retient surtout que l'exécution est *beaucoup plus rapide* tout en restant flexible et fiable. Sa réalisation matérielle (en FPGA par exemple) est assez simple et le principe de base est extensible à des largeurs de mots arbitrairement grandes.

Le CRC16-64 a quand même des inconvénients importants, essentiellement la taille du code, la compaction (que nous verrons dans un prochain article), ainsi que la lourdeur des variables 64 bits pour les petits processeurs 32 bits (tels ARM, MIPS32, ou simplement 80486).

L'aventure ne fait que commencer car toutes les techniques développées dans cet article seront réutilisées pour concevoir un remplaçant du CRC16. Le prochain algorithme sera moins rapide que le CRC16-64, mais si peu gourmand qu'il n'a plus besoin de tables en mémoire. Donc idéal pour les applications embarquées :-)

Je remercie vivement les organisations et les nombreuses personnes qui m'ont aidé à valider le code de cet article, en faisant tourner les auto-tests et en m'envoyant les résultats commentés. Merci aussi à Benoît-Pierre pour son soutien logistique indispensable ! Et si vous trouvez que cet article est alambiqué, attendez de lire les suivants.

LIENS

- Miroir des sources : http://ygdes.com/sources/
- Le musée vivant et allemand des super-ordinateurs : http://www.cray-cyber.org
- Testez gratuitement les ordinateurs HP : http://www.testdrive.hp.com (Alpha, Itanium, HP-PA, Xeon, Opteron...)
- Les CRC expliqués par Terry Ritter dans Dr Dobb's : http://www.ciphersbyritter.com/
- Et sur comp.arch : http://www.ciphersbyritter.com/crchash.htm (voir en particulier le quiproquo sur la configuration d'un registre de CRC)
- Les innombrables et incroyables extensions de GCC : http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/C-Extensions.html
- La syntaxe qui force une variable à résider dans un registre particulier : http://gcc.gnu.org/onlinedocs/gcc-3.4.3/gcc/Local-Reg-Vars.html
- COHEN (Danny), « *On holy wars and a plea for peace* », 1er avril 1980 : http://www.ietf.org/rfc/ien/ien137.txt
- CRAWFORD (Michael D.), « *Porting is good for code* », 2002 : http://www.goingware.com/tips/getting-started/

Yann Guidon,

whygee@f-cpu.org

Électronique, musique et informatique en folie^Wliberté

http://ygdes.com

Annexe : code source de la fonction de signature de la partie 3.5

```
volatile U64 moins_un= (U64)-1LL; /* pour les masques */
void unaligned_CRC16_64(U32 taille_du_bloc, U8 *q) {
/* attention : changement de la taille des accès,
 de U64 à U8, car la granularité d'accès est maintenant
 d'un octet (sinon gcc vire les 3 LSB pour aligner) */
/* utilise 6 entiers 64 bits (+2 temporaires) */
  U64 r0, r1, r2, r3; /* variables de calcul du CRC */
  U64 mask, t; /* pour construire les masques */
/* 4 variables + 2 paramètres d'appel
 = 6 registres 32 bits (quasi-limite du x86)  */
  unsigned offset = PTR_CAST q & 7; /* alignement */
    signed rem;  /* nombre d'octets restant dans le mot */
  U64 *ptr_table = CRC16_64_const + (offset<<4);
  int index; /* peut utiliser
      le même registre que ptr_table */
  /* Recopie la bonne partie de la table des constantes
  memcpy(CRC16_64_table,
       CRC16_64_table_rotations+(offset<<7),128);
    problème : memcpy est "blocant" donc on "déroule"
    et "entrelace" (saupoudre) les instructions.
  Technique expliquée dans GLMF de février 2007.
  Lit en premier les données qui nous intéressent : */
    r0 = ptr_table[0];  CRC16_64_table[0] = r0;
    r1 = ptr_table[1];  CRC16_64_table[1] = r1;
  /* puis remplit le pipeline avec plein d'instructions,
  pour "masquer" la latence d'exécution des autres.
  On compte sur le renommage des registres par le CPU
  pour compenser le codage sub-optimal ci-dessous : */
```

```
  /* reste de l'initialisation */
  mask=moins_un;
    r2 = ptr_table[14];  CRC16_64_table[14] = r2;
  rem = taille_du_bloc - 8; /* décompte un mot entier */
    r3 = ptr_table[15];  CRC16_64_table[15] = r3;
/* Problème lors de la copie : lorsque les données ne
   sont pas touchées ou réutilisées dans un calcul,
   gcc3.3.4 passe stupidement par une paire de registres
   32 bits, au lieu d'utiliser un registre MMX pour
   déplacer les valeurs 64 bits. Une petite macro
   fait l'affaire. Et pour booster, une version SSE
   (128bits) est aussi proposée : */
#if defined (__SSE__)  /* version 128 bits */
  #define MOVTABLE(index) {                  \
     __builtin_ia32_storeups                 \
          ((float*)(CRC16_64_table+index) , \
          __builtin_ia32_loadups             \
          ((float*) (ptr_table+index)));  }
#else
  #if defined (__MMX__) /* version 64 bits forcée */
    /* car aucune intrinsèque pour movq disponible */
    #define MOVTABLE(index) {      \
      register U64 y asm("mm0");   \
      register U64 z asm("mm1");   \
      y = ptr_table[index];        \
      z = ptr_table[index+1];      \
      CRC16_64_table[index]   = y; \
      CRC16_64_table[index+1] = z; }
  #else /* version normale */
    #define MOVTABLE(index) {                     \
      CRC16_64_table[index]   = ptr_table[index];   \
      CRC16_64_table[index+1] = ptr_table[index+1]; }
  #endif
#endif
    MOVTABLE(2);
  /* aligne le pointeur sur 8 octets : */
  q = (U8*) ((PTR_CAST q) & -8L);
    MOVTABLE(4);
  /* création du masque pour traiter le premier mot */
  mask = LOCAL_SHLr(mask, (offset<<3));
    MOVTABLE(6);
  /* lecture des 8 premiers octets (alignés) */
  t = *(U64*)q;
  q+=8;
    MOVTABLE(8);
  /* calcule un pas de CRC avec t comme donnée */
  t  = U64_XOR( U64_XOR(r0,t), U64_XOR(r1,r2));
    MOVTABLE(10);
  /* cas où trop peu d'octets sont à traiter */
  rem += offset; /* compense avec l'offset */
    MOVTABLE(12);
  if (rem <= 0) {  /* version court-circuit */
    rem = (-rem) << 3;
    /* Il faut encore tronquer le masque.
       Deux décalages en sens opposés sont plus
       simples qu'un masque supplémentaire. */
    #if defined (__MMX__) /* version MMX optimisée */
    {
      U64 tmp;
      U64_MOVD(tmp, rem)
      mask = U64_SHR(U64_SHL(mask, tmp), tmp);
    }
    #else
      mask = LOCAL_SHR(LOCAL_SHL(mask, rem), rem);
    #endif
   /* réécrit uniquement les octets choisis */
    t    = U64_AND(t,mask);
    mask = U64_ANDN(mask, r0);
    r0   = U64_OR(mask, t) ;
    CRC16_64_table[0] = r0;
   /* à ce moment, on pourrait tenter une
    "compaction rapide" avec des valeurs précalculées */
  }
```

```
 /* cas normal où plus d'un mot doit être traité */
 else {
  /* note : ce code est une copie du précédent bloc,
          pour réduire le nombre de branchements */
    t    = U64_AND(t,mask);
    mask = U64_ANDN(mask, r0);
    r0   = U64_OR(mask, t) ;
    CRC16_64_table[0] = r0;
  /* la version ultra-rapide */
    #define CRC_ROUND_U8(x, a, b, c)                  \
       r##b = CRC16_64_table[(x+1) & 15];             \
       r##a = U64_XOR( U64_XOR( r##a , ((U64*)q)[x-1]), \
                       U64_XOR( r##b , r##c ));       \
       CRC16_64_table[(x) & 15] = r##a ;
    while (rem >= 128) {
 /* l'instanciation : a = (x) % 4
                    |  b = (x+1) % 4
                    |  |  c = (x+14) % 4  */
      CRC_ROUND_U8( 1, 1, 2, 3)
      CRC_ROUND_U8( 2, 2, 3, 0)
      CRC_ROUND_U8( 3, 3, 0, 1)
      CRC_ROUND_U8( 4, 0, 1, 2)
      CRC_ROUND_U8( 5, 1, 2, 3)
      CRC_ROUND_U8( 6, 2, 3, 0)
      CRC_ROUND_U8( 7, 3, 0, 1)
      CRC_ROUND_U8( 8, 0, 1, 2)
      CRC_ROUND_U8( 9, 1, 2, 3)
      CRC_ROUND_U8(10, 2, 3, 0)
      CRC_ROUND_U8(11, 3, 0, 1)
      CRC_ROUND_U8(12, 0, 1, 2)
      CRC_ROUND_U8(13, 1, 2, 3)
      CRC_ROUND_U8(14, 2, 3, 0)
      CRC_ROUND_U8(15, 3, 0, 1)
      CRC_ROUND_U8(16, 0, 1, 2)
 /* note : c'est décalé par rapport à la version
 originale car la première ronde est déjà calculée
 mais ça décale l'accès à la mémoire, d'où le (x-1)
 dans la macro */

      rem -= 128;
      q += 128;
    }
    index=1;
    /* la version... juste rapide */
    while (rem >= 8) {
      CRC16_64_table[index] = U64_XOR(
        U64_XOR(CRC16_64_table[index], *(U64*)q),
        U64_XOR(CRC16_64_table[(index+1) &15],
                CRC16_64_table[(index+14)&15]));
      index=(index+1) & 15; /* rotation des registres */
      rem -= 8;
      q += 8;
    }
/* finalement, on traite le petit bout qui dépasse : */
    if (rem != 0) {
      r0 = CRC16_64_table[index];
      /* dernier calcul */
      t = U64_XOR(
          U64_XOR(r0 , *(U64*)q),
          U64_XOR(CRC16_64_table[(index+1) &15],
                  CRC16_64_table[(index+14)&15]));
      /* construction du nouveau masque */
      mask = moins_un;
      rem <<= 3;
      mask = LOCAL_SHLr(mask, rem);
      /* sélectionne les bons octets */
      r0   = U64_AND(r0,mask);
      mask = U64_ANDN(mask, t);
      r0   = U64_OR(mask, r0) ;
      CRC16_64_table[index] = r0;
    }
  }
}
#undef MOVTABLE
#undef CRC_ROUND_U8 /* petit nettoyage */
```

Stephane Ducasse & Serge Stinckwich

Introspection et méta-manipulations en Squeak

Squeak, comme tous les Smalltalk, est un langage de programmation réflexif. Être réflexif pour un langage de programmation signifie permettre, d'une part, l'introspection, c'est-à-dire autoriser l'analyse des structures de données qui définissent le langage lui-même comme les objets, les classes, la pile d'exécution et, d'autre part, l'intercession, c'est-à-dire permettre de modifier depuis le langage lui-même sa sémantique et son comportement, comme ajouter des méthodes espionnes, changer la pile d'exécution pour construire un débogueur. Notons que bien que Java définisse une interface réflexive, il n'est pas totalement réflexif et offre seulement une introspection limitée. Dans cet article, nous montrons avec quelques exemples des aspects introspectifs de Squeak et nous aborderons dans un prochain article l'intercession. Notez que c'est cette particularité qui fait que le framework web Seaside, que nous avons présenté dans un article précédent, existe en Smalltalk. Nous avons choisi des exemples pratiques qui peuvent vous aider dans votre programmation quotidienne.

Ouvrons le couvercle : les instances

Lors d'un précédent article montrant comment définir une classe, nous avons vu comment utiliser un inspecteur. Un inspecteur est un outil de développement qui permet non seulement de voir et modifier les valeurs des variables d'instances d'un objet, mais également de lui envoyer des messages. Exécutez dans un *workspace* le code qui suit :

```
| w |
w := Workspace new.
w openLabel: 'myworkspace'.
w inspect
```

Vous obtenez un inspecteur qui, dans la partie gauche, présente les variables d'instances (ici `dependents`, `contents` et `bindings`) et la valeur de la variable sélectionnée dans la partie droite. Si vous sélectionnez la variable `contents` qui représente le texte contenu dans le workspace, vous obtenez une chaîne vide. Maintenant tapez `'1+2'` à la place de la chaîne vide, puis faites `accept`. La valeur de la variable `contents` a changé. Pour voir son effet dans le workspace évaluer (*do it*) `self contentsChanged` dans la partie basse de l'inspecteur, car l'accesseur `contents:` ne propage pas ses changements automatiquement.

Comment l'inspecteur fonctionne-t-il ? En Smalltalk, les variables d'instances sont protégées. Il est donc théoriquement impossible d'y accéder si la classe ne définit pas d'accesseurs. Néanmoins, l'inspecteur permet d'accéder à n'importe quelle variable d'instances. Il utilise pour cela les possibilités réflexives de Smalltalk. L'inspecteur est basé sur le fait que l'on peut accéder à la valeur d'une variable d'instance grâce aux méthodes `instVarAt:`, `instVarAt:put:`, `instVarNamed:`, `instVarNamed:put:` définies sur la classe `Object` :.

- `instVarAt: unEntier` permet d'accéder à la variable d'instance de position `unEntier` ;
- `instVarAt: unEntier put: uneValeur` permet de modifier la variable d'instance de position `unEntier` ;
- `instVarNamed: uneChaine` permet d'accéder à la variable d'instance définie par `uneChaine` ;
- `instVarNamed: uneChaine put: uneValeur` permet de changer la valeur de la variable d'instance représentée par `uneChaine`.

Par exemple, on peut modifier directement la valeur de la variable `contents` du workspace `w` par :

```
w instVarNamed:'contents' put:'3+4'; contentsChanged.
```

Notons que ces méthodes sont utiles pour construire des outils de développement ou de débogage, mais leur utilisation lors de développement habituel d'applications est à proscrire, car elles violent les principes d'encapsulation promues par les langages à objets.

Les méthodes `instVarAt:` et `instVarAt:put:` sont des méthodes primitives. Si vous consultez le code de ces méthodes, vous verrez qu'elles appellent une opération primitive de la machine virtuelle de Squeak, en utilisant la syntaxe très particulière : `<primitive: xx>` où `xx` est un entier. Ces opérations sont en nombre limité et constituent en quelque sorte les actions de base de Squeak : opérations de base sur les entiers, accès aux périphériques d'entrées-sorties (disque, écran, réseau...). Certaines primitives permettent également d'accélérer des fonctions du système. Il est également possible de définir ses propres primitives et de les relier à des *plugins* écrits en C afin de bénéficier de toute la rapidité de ce langage depuis Squeak.

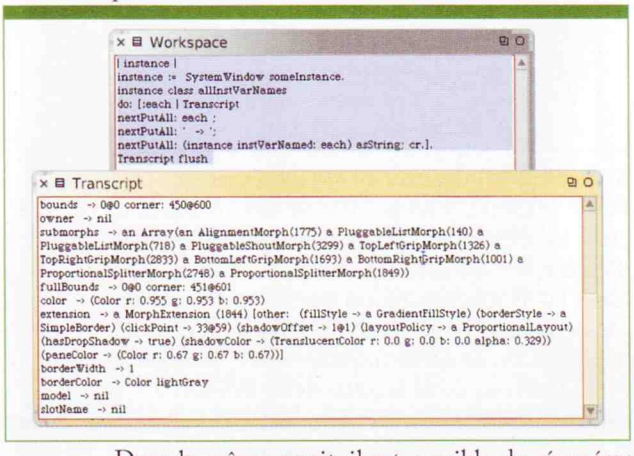
Voici le code de la méthode `instVarAt:` de la classe `Object` :

```
instVarAt: index
"Primitive. Answer a fixed variable in an object. The numbering of
the variables corresponds to the named instance variables. Fail if
the index is not an Integer or is not the index of a fixed variable.
Essential. See Object documentation whatIsAPrimitive."
<primitive: 73>
"Access beyond fixed variables."
^self basicAt: index - self class instSize
```

Le code écrit après la primitive n'est pas exécuté en temps normal. Il n'est exécuté que lorsque la primitive échoue, par exemple ici si on essaie d'accéder à une variable d'instance qui n'existe pas.

Ce qui permet d'avoir le débogueur Smalltalk qui se déclenche même sur les primitives. Il est tout à fait possible de modifier les méthodes contenant des primitives. Mais attention, ceci est très dangereux pour la stabilité de votre système.

Le script suivant exécuté dans un espace de travail (workspace) montre comment on peut afficher dans la fenêtre du Transcript (menu *open ... Transcript*) le contenu des variables d'instances d'une instance de la classe `SystemWindow` sélectionnée au hasard. La méthode `allInstVarNames` permet d'obtenir toutes les variables d'instances d'une classe. La méthode `someInstance` parcourt la mémoire et retourne la première instance d'une classe.



Dans le même esprit, il est possible de récupérer des instances qui vérifient certaines propriétés. Par exemple, l'expression suivante permet d'obtenir toutes les instances de la classe `Browser` dont la variable d'instance `systemOrganizer` n'est pas `nil` :

```
Browser allInstances select: [ :c | (c instVarNamed:
'systemOrganizer') notNil]
```

Regardons l'implantation de la méthode `instanceVariableValues` définie sur la classe `Object` qui rend un tableau dont les éléments sont les valeurs des variables d'instances définies exclusivement sur la classe (et non les variables d'instances héritées). Cet exemple montre l'utilité de pouvoir accéder aux variables via leur position. La méthode `intSize` retourne le nombre de variables d'instances d'une classe.

`(1@2) instanceVariableValues` retourne `anOrderedCollection(1 2)`.

```
Object>>instanceVariableValues
  "Answer a collection whose elements are the values of those instance variables
  of the receiver which were added by the receiver's class"
  | c |
  c := OrderedCollection new.
  self class superclass instSize + 1
  to: self class instSize
  do: [:i | c add: (self instVarAt: i)].
 ^ c
```

On parcourt l'ensemble des variables d'instances définies par la classe en utilisant les positions des variables d'instances (de la fin des variables introduites dans les superclasses aux variables de cette classe).

L'ensemble des outils de développements de Smalltalk (*browser* de code, inspecteur...) utilise de manière très conséquente ces possibilités introspectives. Voici d'autres méthodes pouvant servir lors de la construction d'outils de développement :

- `isKindOf: uneClasse` rend vrai si le receveur est instance de la classe ou de la superclasse spécifiée. Par exemple `1.5 isKindOf: Number` rend `true` alors que `1.5 isKindOf: Integer` rend `false`.
- `respondsTo: unSymbol` rend vrai si le receveur sait exécuter la méthode spécifiée par `unSymbol`. Par exemple, `1.56 respondsTo: #floor` rend `true`, car la classe `Number` implante la méthode `#floor` qui arrondit un nombre. `Exception respondsTo: #`, rend `true` ce qui signifie que l'on peut créer un ensemble d'exceptions en envoyant le message `#`, à la classe `Exception`.

Encore une fois ces fonctionnalités ne sont intéressantes que lors de la définition d'outils de programmation et leur utilisation est dangereuse pour le développement d'applications. En effet, utiliser le type d'un objet pour conditionner l'exécution de méthodes est souvent un signe de problèmes de conception. Smalltalk offre d'autres possibilités d'introspection sur des objets tels que les méthodes, la pile d'exécution qui sont représentées par des objets seulement à la demande, les processus, le gestionnaire de mémoire... Mais nous ne pouvons pas tout aborder ici, essayez de lire le code des classes et expérimentez par vous-même.

Naviguer dans le code

Nous avons déjà vu précédemment qu'avec Smalltalk, les classes sont des objets comme les autres. Elles offrent de nombreuses fonctionnalités très intéressantes dont voici quelques exemples. Comme montré par les exemples précédents, il est possible d'obtenir une instance quelconque d'une classe en envoyant le message `someInstance` à une classe. Il est aussi possible d'obtenir toutes les instances d'une classe à l'aide du message `allInstances`, ainsi que le nombre d'instances en mémoire en utilisant le message `instanceCount`. Ces fonctionnalités sont de très puissants outils pour déboguer une application par exemple :

```
Morph instanceCount.
```

La valeur retournée va dépendre du nombre d'instances de Morph que vous utilisez à l'instant présent.

Regardons maintenant comment les classes en Smalltalk offrent de nombreuses possibilités de références croisées et navigation entre méthodes et classes. Voici quelques exemples qui montrent à quel point les classes permettent de construire facilement des outils de navigation et d'analyse :

- `Point whichSelectorsAccess: 'x'` retourne un ensemble de sélecteurs :

```
an IdentitySet(#rotateBy:about: #translateBy:
#isInsideCircle:with:with: #sideOf: #nearestPointAl
ongLineFrom:to: #normalized #eightNeighbors
#dist: #hash #rotateBy:centerAt: #theta #grid:
#fourNeighbors #dotProduct: #scaleFrom:to: #normal
#onLineFrom:to:within: #+ #degrees #interpolateTo:at:)
```

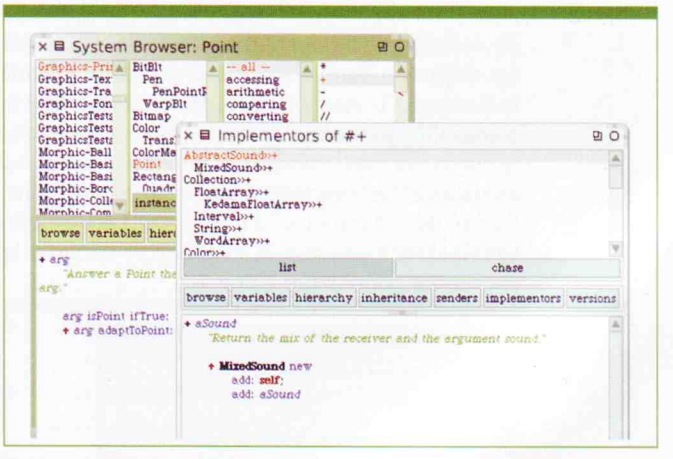
C'est-à-dire la liste des méthodes qui accèdent en lecture à la variable d'instance `x` de la classe `Point`.

- `Point whichSelectorsStoreInto: 'x'` rend la liste de méthodes qui accèdent en écriture à la variable d'instance x de la classe `Point`.
- `Point whichSelectorsReferTo: #+` rend la liste de méthodes définies sur la classe `Point` qui invoque la méthode `+`.
- `Point crossReference` retourne un tableau où à chaque méthode de la classe est associé un ensemble de messages invoqués par elle.
- Rectangle `whichClassIncludesSelector: #inspect` rend la classe qui définit le message `inspect`, c'est-à-dire ici la classe `Object`.
- Rectangle `unreferencedInstanceVariables` rend la liste des variables d'instances qui ne sont pas référencées dans la classe ou ses sous-classes (ici vide).

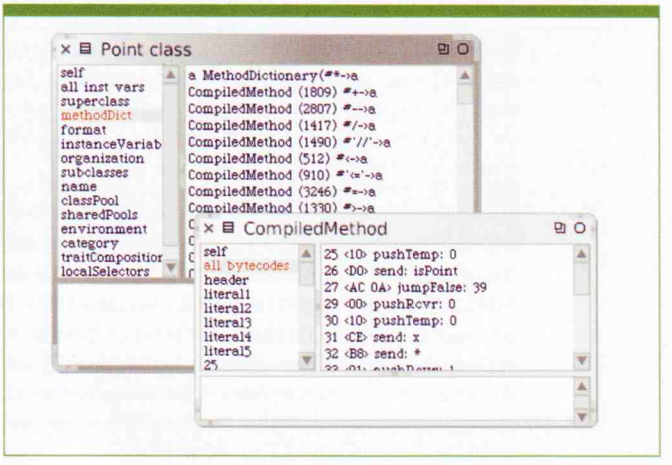
La classe `SystemNavigation` permet de parcourir facilement les classes définies dans une image Squeak :

- `SystemNavigation default` permet d'accéder à l'instance de navigation par défaut.
- `SystemNavigation default allClassesImplementing: #+.` retourne toutes les classes implémentant le message `#+`. On obtient ici : `an Array(Number Fraction Float Integer SmallInteger LargePositiveInteger ScaledDecimal DateAndTime Duration Timespan Player Color Collection WordArray FloatArray KedamaFloatArray String Interval AbstractSound MixedSound Point Voice Complex TraitDescription TraitComposition TraitTransformation TcomposingDescription).` On voit donc qu'il est possible d'additionner non seulement 2 entiers, mais également 2 fractions, 2 chaînes de caractères ou encore 2 intervalles.
- `SystemNavigation default allSentMessages.` retourne tous les messages envoyés dans l'image Smalltalk (plus de 26000 messages dans l'image que nous utilisons).
- `SystemNavigation default allUnSentMessages.` retourne tous les messages implémentés dans le système, mais envoyés par aucun objet (plus de 6000 messages...) dans l'image. Ces messages ne sont pas nécessairement inutiles.
- `SystemNavigation default allUnimplementedCalls.` retourne tous les messages envoyés, mais non implémentés. C'est là plus grave : il y a des fonctionnalités qui ne sont pas réalisées dans l'image.
- `SystemNavigation default allCallsOn:#Point.` retourne toutes les références sur la classe `Point`.

Ces fonctionnalités sont intégrées dans l'environnement de programmation de Squeak, notamment dans les navigateurs de classes. Il est possible, par exemple en sélectionnant une méthode, de connaître les autres méthodes de même nom définies dans d'autres classes (*implementors*) ou bien la liste des méthodes qui utilisent cette méthode (*senders*). Ceci s'avère très pratique, dès qu'il s'agit par exemple de modifier le nom d'une méthode afin de s'assurer qu'il n'y a plus d'appels à cette méthode.



Si on utilise un inspecteur sur une classe (par exemple : `Point inspect`), il est possible de voir comment la classe est réalisée sous la forme d'un objet. Par exemple dans la figure suivante, on inspecte la structure de la classe `Point`. On remarquera que les méthodes sont conservées dans un dictionnaire indexé par le nom des méthodes. Un deuxième inspecteur a été ouvert sur le *bytecode* décompilé de la méthode `#*` de `Point`.



Avec Smalltalk, on a vraiment un système complètement ouvert et transparent, où tout le code est accessible en permanence. L'utilisateur peut étudier tout le système comme il le souhaite, car il n'y a pas de code caché. La portion primitive du système (c'est-à-dire qui n'est pas écrite en Smalltalk) est réduite à sa plus simple expression. En Squeak, même la machine virtuelle est écrite en Smalltalk, mais pour avoir des performances intéressantes, cette machine virtuelle est traduite en C et compilée sous la forme d'un exécutable. Il est néanmoins possible pour déboguer la machine virtuelle de Squeak de l'exécuter dans Squeak ! Cela marche, mais c'est très lent.

Applications pratiques de l'introspection

Métriques de code

Voyons maintenant un exemple concret d'utilisation des capacités introspectives de Smalltalk pour rapidement définir des métriques de code. Une métrique de code est une mesure que l'on peut faire sur du code telle que : la profondeur de l'héritage, le nombre de méthodes de chaque classe, le nombre de variables d'instances, le nombre de méthodes définies localement, le nombre de variables d'instances ajoutées localement, le nombre de sous-classes, et le nombre total de sous-classes. Lorsque l'on aborde une application inconnue, calculer des métriques permet d'obtenir une première idée de l'application. Les expressions suivantes illustrent comment ces métriques peuvent être calculées :

```
profondeurHeritage := Morph allSuperclasses size. => 2
nbMethodes := Morph allSelectors size. => 1593
nbInstances := Morph allInstVarNames size.  => 6
nbMethodesAjoutees := Morph selectors size.  => 1165
nbInstanceAjoutees := Morph instVarNames size.  => 6
nbSousclasses := Morph subclasses size.  => 45
nbTotalSousclasses := Morph  allSubclasses size. => 412
```

Les chiffres peuvent varier en fonction de l'image Squeak que vous utilisez. Les résultats pour la classe `Morph` (classe de tout objet graphique de Squeak) nous montrent qu'il s'agit d'une classe disposant d'un nombre très important de sous-classes et un nombre conséquent de méthodes. Cette classe nécessiterait des refactorisations !

Une des métriques les plus intéressantes dans le domaine des langages à objet est de connaître le nombre de méthodes qui étendent des méthodes héritées de leur superclasse. Cela permet notamment d'avoir une meilleure compréhension de la relation qui existe entre une classe et ses superclasses.

Voici un script montrant comment identifier les méthodes de la classe `Browser` qui font un appel à une méthode cachée, c'est-à-dire une invocation via `super` de la forme suivante :

```
Browser>>xx
          ...
          super xx

Browser selectors select: [:eachSelector |
          | method |
          method := Browser compiledMethodAt: eachSelector.
          method sendsToSuper]
```

retourne comme résultat une liste de méthodes, dont, par exemple, la méthode `#veryDeepInner:`. Dans le code qui précède, on demande à la classe `Browser` l'ensemble des messages qu'elle comprend (*selectors*) et on sélectionne parmi ceux-ci, ceux dont le *bytecode* de la méthode correspondante contient un appel à la super-classe.

On voit ici que le bytecode Smalltalk est lui-même considéré comme un objet auquel on peut s'adresser pour avoir des informations. On pourrait même imaginer des méthodes qui transforment ce code.

Détection de possibles erreurs

Invoquer une méthode masquée via la variable `super` en utilisant un nom de méthode différent que celui de la méthode contenant `super` peut créer des bugs lorsque la classe est sous-classée. Tout bon développeur évite ce genre de code. Identifier de telles pratiques peut être très simplement réalisé. Il suffit de trouver les méthodes effectuant un appel via `super` qui ne contiennent pas le sélecteur de la méthode analysée :

```
Collection selectors
          select: [:eachSelector | | method |
                    method := Collection
compiledMethodAt: eachSelector.
                    method sendsToSuper and: [(method
messages includes: eachSelector) not]].
```

Notons que ce script ne détecte pas toutes les méthodes. Ainsi, les méthodes contenant des boucles ou des récursions comme ci-dessous ne sont pas détectées :

```
xxx
super yyy.
self xxx.
```

En exécutant ce script sur la classe `Collection`, vous devez obtenir la méthode `printNameOn:` qui effectivement invoque `printOn:` au moyen de `super` :

```
Collection>>printNameOn: aStream
super printOn: aStream
```

et comme on a la méthode :

```
Collection>>printOn: aStream
"Append a sequence of characters that identify the
receiver to aStream."
self printNameOn: aStream.
self printElementsOn: aStream
```

la méthode `printOn:` de `Collection` pourrait être refactorisée de la manière suivante :

```
Collection>>printOn: aStream
"Append a sequence of characters that identify the
receiver to aStream."
self printOn: aStream.
self printElementsOn: aStream
```

et la méthode `printNameOn:` supprimée. La méthode `printNameOn:` est-elle utilisée par d'autres classes ? Demandons à Squeak :

```
SystemNavigator default allCallsOn:#printNameOn:
retourne :
an OrderedCollection(a MethodReference RunArray >>
printOn: a MethodReference Bitmap >> printOn: a
MethodReference Text >> printOn: a MethodReference
CompiledMethod >> printOn:)
```

Il faudrait donc également refactoriser les classes `Bitmap`, `RunArray`, `CompiledMethod` et `Text` qui contiennent des appels à la méthode `printOnName:`.

Des points d'arrêts intelligents

Dans la version 3.8 de Squeak, une nouvelle forme de points d'arrêt très utiles a été introduite avec la méthode `haltIf:` de la classe `Object`. Ce point d'arrêt stoppe l'exécution d'un programme seulement si la

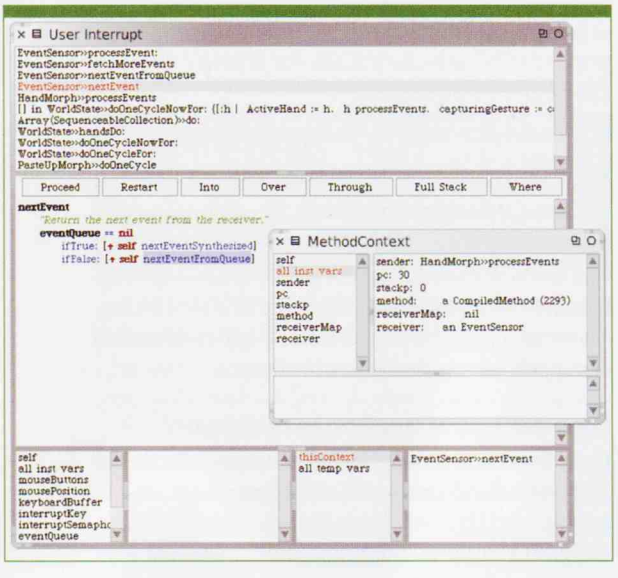
méthode a été appelée par une méthode dont le nom est passé en argument. Cela est très pratique pour arrêter une méthode seulement lors de l'exécution d'un test et pas à chaque fois que la méthode est exécutée. Ainsi, l'expression `self haltIf: #testFoo` n'arrêtera la méthode le contenant exclusivement si cette méthode est invoquée directement ou indirectement depuis la méthode `testFoo`. La définition de la méthode `haltIf:` est assez simple et tient en cinq lignes. Elle utilise la variable `thisContext` qui permet d'accéder à la pile d'exécution sous forme d'un objet (`thisContext` est l'un des 6 mots clés de Smalltalk avec `self`, `super`, `nil`, `true`, `false`).

```
Object>>haltIf: condition
    "This is the typical message to use for inserting breakpoints during
    debugging.  Param can be a block or expression, halt if true.
    If the condition is a selector, we look up in the callchain. Halt if
    any method's selector equals selector."

    | cntxt |
    cntxt := thisContext.
    [cntxt sender isNil] whileFalse: [
          cntxt := cntxt sender.
          (cntxt selector = condition) ifTrue: [Halt signal]].
    ^self.
```

Ensuite, elle remonte la pile d'exécution et regarde si le nom de la méthode appelante est celui donné en paramètre. Si c'est le cas, elle lève une exception qui par défaut conduit à l'ouverture du débogueur. Cet exemple montre toute la puissance de Smalltalk qui permet de définir des fonctionnalités et des outils très puissants depuis le langage lui-même.

La pseudo-variable `thisContext` est particulièrement utilisée dans le débogueur de Squeak. Il s'agit d'une instance de la classe `MethodContext`. Elle contient des informations sur la méthode en cours d'exécution, la méthode, le pointeur de pile (`stackpc`) et pointeur de programme (`pc`).

Dans l'exemple ci-dessous, on a stoppé la boucle d'évaluation de Squeak, au moyen de [ALT-.] et inspecté le contexte courant.



Seaside, dont nous avons parlé dans un article précédent, utilise également la variable `thisContext` pour accéder à la pile d'exécution du programme et la modifier en cours de route afin d'implémenter facilement des composants réutilisables au-dessus de HTTP.

Conclusion

L'utilisation des capacités introspectives est omniprésente et naturelle en Smalltalk. En effet, tout l'environnement de développement et ses innombrables navigateurs de code sont construits en utilisant les interfaces d'introspection des objets et des classes. Il faut remarquer que pouvoir accéder aux objets qui représentent le programme est très utile : ceci évite de devoir construire une autre représentation comme des arbres de syntaxe abstraits pour construire des environnements de programmation (Eclipse). C'est pour réparer cette lacune en la matière de la version 1.0 de Java que Java 1.2 a vu apparaître l'API « réflexive » (qui n'est qu'introspective). De plus, le fait que les classes et autres objets qui assurent l'exécution même des programmes offrent des interfaces introspectives assure que les représentations des programmes sont toujours synchronisées avec le code. Ainsi, il n'est pas nécessaire de maintenir ces représentations à jour.

Dans un prochain article, nous vous montrerons comment les aspects réflexifs servent pour prototyper très rapidement des applications. Nous montrerons aussi comment le contrôle de l'envoi de message permet d'implanter certains *design patterns* comme le Proxy et de créer des outils d'espionnage des objets.

Pour cet article, nous vous invitons à utiliser l'image Squeak préparée par Damien Cassou. Cette image remise à jour très régulièrement (la dernière version courante est la 83) contient de très nombreux outils utiles pour les développeurs sans avoir besoin de les installer (Shout, OmniBrowser, RoelTyper, ECompletion, DynamicProtocols…).

Stephane Ducasse & Serge Stinckwich,

stephane.ducasse@gmail.com
Serge.Stinckwich@gmail.com

LIENS

- Le site officiel : http://www.squeak.org/
- Le wiki de la communauté : http://wiki.squeak.org/Squeak
- Le wiki de la communauté française : http://community.ofset.org/wiki/Squeak
- Image Squeak-dev de Damien Cassou : http://damien.cassou.free.fr/squeak-dev/
- Le groupe des utilisateurs européens de Smalltalk (*European Smalltalk User Group*). L'adhésion est gratuite : http://www.esug.org/
- Des livres gratuits en ligne sur Smalltalk et Squeak : http://www.iam.unibe.ch/~ducasse/FreeBooks.html
- Un livre sur Squeak en français : http://www.iam.unibe.ch/~ducasse/Books.html : BRIFFAULT (X.) ET DUCASSE (S.), *Squeak*, Eyrolles, 2002.

Raphaël Semeteys

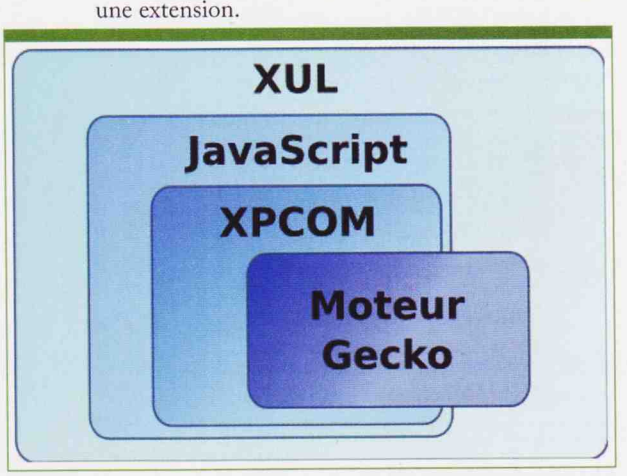
Développer une première extension Firefox

L'objectif de cet article est de vous initier à l'écriture d'extensions pour le navigateur Mozilla Firefox. Il n'a ni la prétention d'être exhaustif ni d'être très détaillé, mais plutôt de vous donner une vue d'ensemble et d'introduire les techniques, les outils et les bonnes pratiques.

La théorie

Les différents composants et concepts en jeu

Avant d'entrer plus dans les détails dans la construction de notre extension, il est nécessaire de lister et de positionner les différents composants de Firefox à considérer. Cette synthèse nous permettra également de déterminer quels sont les concepts à connaître et les compétences techniques requises pour développer une extension.



Le moteur Gecko

Gecko est le moteur d'affichage de Firefox. Il implémente notamment les standards suivants : X/HTML, CSS, DOM, JavaScript, XML, XSL, XPath, RDF et SVG. Il est également responsable de l'affichage des pages XUL (*XML-based User interface Language*) qui servent à décrire l'IHM de Firefox et de ses extensions (le langage XUL est abordé en détail plus loin).

Composants XPCOM

XPCOM (*Cross-Platform Component Object Model*) est le modèle de composants de Mozilla, similaire à CORBA ou Microsoft COM. Ce modèle permet de développer des composants dans différents langages (C++, C, JavaScript, Python et d'autres) et de les faire communiquer. La plupart des fonctionnalités du moteur Gecko sont visibles sous forme de composants XPCOM.

La couche d'interface graphique

Langage XUL

XUL (*XML User Interface Language*), qui se prononce « zoul », est un langage de description d'interfaces graphiques, basé sur le standard XML. Il s'utilise conjointement avec CSS, RDF, DOM et JavaScript. Il permet de décrire et de positionner en XML des composants graphiques ou *widgets* (fenêtres, boutons, listes, menus, zones d'édition, etc.), de leur associer des évènements, des styles et des sources de données, ainsi que de les modifier via des scripts. Les développeurs habitués au DHTML n'auront pas de problème à s'adapter à XUL.

Bindings XBL

Firefox supporte également le langage XBL (*XML Bindings Language*) qui permet de lier aux éléments d'un document XML, des comportements, des interfaces ou des modèles de contenu. Couplé avec XUL, il est utilisé pour étendre et personnaliser les widgets.

Code JavaScript

Mozilla Firefox embarque le moteur SpiderMonkey capable d'interpréter et d'exécuter des instructions JavaScript. Bien qu'elles puissent elles-mêmes contenir des composants XPCOM, les extensions sont généralement codées en langage JavaScript. Elles utilisent la technologie XPConnect (*Cross Platform Connect*) pour manipuler les composants XPCOM de Firefox et accéder aux fonctionnalités du navigateur. Le code JavaScript fait donc le lien entre l'interface graphique décrite en XUL et les composants XPCOM de Gecko.

Chrome et la sécurité

Les fichiers XUL, lorsqu'ils sont accédés via le protocole HTTP, sont soumis aux mêmes restrictions de sécurité que les pages HTML. De façon à lever ces restrictions, le moteur Gecko gère un référentiel dans lequel peuvent être enregistrés les fichiers XUL. Ces fichiers sont alors accessibles via un protocole spécifique et des URL du type `chrome://<paquetage>/<partie>/<fichier.xul>` (par exemple tapez `chrome://browser/content/browser.xul` dans la barre d'adresses de Firefox ;). Du point de vue du développement d'extensions, le terme de « chrome » regroupe à la fois le référentiel et le protocole d'accès aux fichiers XUL.

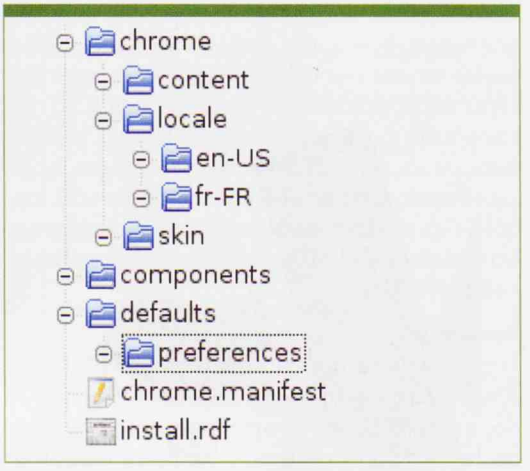
Concept d'overlay

Firefox utilise la notion d'« *overlay* » pour partager des éléments entre plusieurs pages XUL, mais aussi

pour surcharger et étendre un fichier XUL sans pour autant modifier celui-ci. C'est cette méthode qui est utilisée pour insérer les extensions dans l'IHM de Firefox (menu *Outils*, menu contextuel, etc.) sans pour autant impacter le navigateur.

Paquet XPI

Les extensions sont distribuées et installées sous forme de paquets XPI (*Cross-Platform Install*). Cela se prononce « zippy ». Il s'agit d'une archive Zip regroupant tous les fichiers constituant l'extension.



Pour les extensions Firefox de version 1.5 ou supérieure, l'arborescence de cette archive est organisée de la manière suivante :

- `chrome` : répertoire contenant les fichiers propres à l'extension elle-même (le contenu de ce répertoire peut également être compressé dans une archive JAR) ;
  - `content` : fichiers XUL et JavaScript associés ;
  - `locale` : contient un répertoire par langue supportée par l'extension ;
  - `skin` : regroupe les feuilles de style CSS ;
- `components` : éventuels composants XPCOM ;
- `defaults` : fichiers liés au profil utilisateur ;
  - `preferences` : fichiers de préférences, en JavaScript, exécutés au lancement de l'extension ;
- `chrome.manifest` : fichier d'enregistrement de l'extension dans le référentiel chrome de Firefox ;
- `install.rdf` : fichier contenant les informations nécessaires au gestionnaire d'extensions de Firefox.

Les détails de cette arborescence seront illustrés plus loin dans la partie pratique de cet article.

Résumé : compétences nécessaires

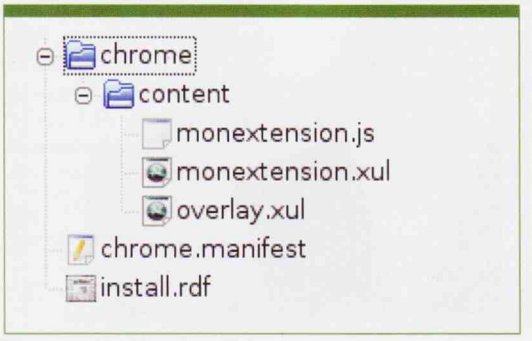
On a vu que Mozilla Firefox utilise de nombreuses technologies et plusieurs langages de programmation. De quoi avoir le vertige lorsque l'on désire réaliser une extension. Le tableau ci-dessous résume le niveau de connaissance qu'il est nécessaire d'avoir pour commencer à développer une extension.

Technologie	Nécessité	Pour quoi faire ?
XUL	Obligatoire	Description de l'IHM de l'extension
JavaScript	Obligatoire	Développement de scripts associés à l'IHM de l'extension
XBL	Optionnel	Définir des composants XUL réutilisables
C, C++	Optionnel	Développement de composants XPCOM
CSS	Optionnel	Définir les styles d'affichage de l'extension
RDF	Optionnel	Définir des modèles de contenus pour des composants XUL

La pratique

Nous allons illustrer la partie théorique précédente en créant une extension et en détaillant chaque étape du processus. L'extension que nous prévoyons d'implémenter est fonctionnellement très limitée, puisqu'elle se borne à afficher une boîte de dialogue affichant un texte d'information à l'utilisateur. L'objectif de cet exemple est donc, vous l'aurez bien compris, uniquement pédagogique.

Dans cet exemple simple, nous nous limiterons à des pages XUL simples, associées à des scripts JavaScript et n'implémenterons ni de composants XBL, ni de composants XPCOM, ni de modèles de contenu RDF. L'arborescence constituant notre extension sera de la forme suivante :



IHM : fichier XUL

Dans le développement de notre première extension, nous allons procéder par itérations : cela nous permettra d'expliquer les différentes étapes et d'introduire les concepts au fur et à mesure.

L'extension sera composée d'une unique page XUL contenant un texte et un bouton. L'appui sur le bouton déclenchera l'apparition d'une fenêtre *popup*. Commençons par décrire l'IHM de notre extension.

Page XUL : `monextension.xul`

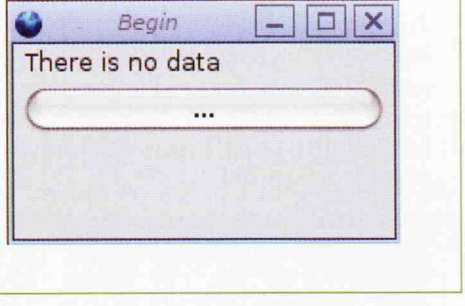
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="chrome://global/skin/"
type="text/css"?>
```

```
<window id="page" title="Begin"
        xmlns="http://www.mozilla.org/keymaster/
gatekeeper/there.is.only.xul"
        width="200px" height="100px">
        <vbox>
                <label value="There is no data"/>
                <button label="..."/>
        </vbox>
</window>
```

Examinons plus en détail le contenu de la page XUL :

- ligne `<?xml version="1.0"?>` : désigne qu'il s'agit d'un fichier XML ;
- ligne `<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>` : import d'une feuille de style par défaut ;
- élément `<window/>` : déclaration d'une fenêtre, en précisant son identifiant, son titre et ses dimensions ;
- l'attribut `xmlns` : déclaration de l'espace de nommage propre au langage XUL ;
- élément `<vbox/>` : boîte qui permet le placement vertical des éléments qu'elle contient ;
- élément `<label/>` : champ texte ;
- élément `<button/>` : bouton (auquel pour le moment aucune action n'a été attachée).

Le résultat affiché par Firefox est le suivant :



Code : JavaScript et XPCOM

Attachons une action à l'évènement « clic sur le bouton ». Pour ce faire, nous utilisons l'attribut `oncommand` de l'élément `<button>` pour appeler une fonction JavaScript.

```
<button label="..." oncommand="show();"/>
```

Il reste à coder la fonction `show` dont le but est d'afficher une fenêtre popup. Nous externalisons le JavaScript dans un fichier `monextension.js`. Pour afficher cette fenêtre, on aurait pu utiliser une deuxième page XUL, mais, pour illustrer l'appel de composant XPCOM, nous allons plutôt utiliser le service *prompt* de Firefox qui permet d'afficher des boîtes de dialogue simples.

Fichier JavaScript : `monextension.js`

```
function show() {
        var prompts = Components.classes["@mozilla.org/embedcomp/
prompt-service;1"]
                .getService(Components.interfaces.nsIPromptService);
        prompts.alert(null, "End", "There is only XUL");
}
```

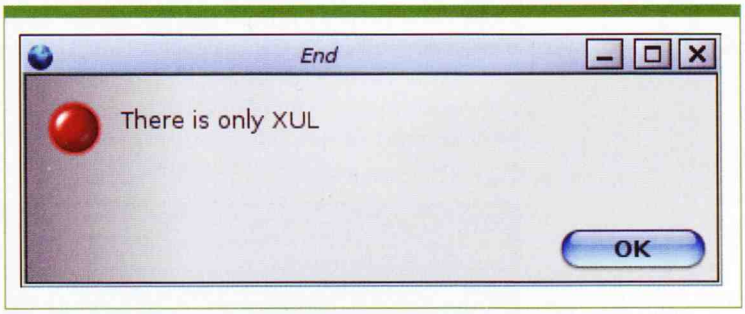
Chaque composant XPCOM est enregistré dans le référentiel de Firefox avec une URI, celle du service prompt est **@mozilla.org/embedcomp/prompt-service;1**. Chaque composant implémente également une ou plusieurs interfaces.

On indique ici, via `Components.interfaces.nsIPromptService` que l'on veut utiliser l'interface `nsIPromptService` du composant en question.

Il nous faut modifier le fichier `monextension.xul` pour y insérer le script :

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window id="page" title="Begin"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
        width="200px" height="100px">

        <script  type="application/x-javascript" src="monextension.js"></script>

        <vbox>
                <label value="There is no data"/>
                <button label="..." oncommand="show();"/>
        </vbox>
</window>
```

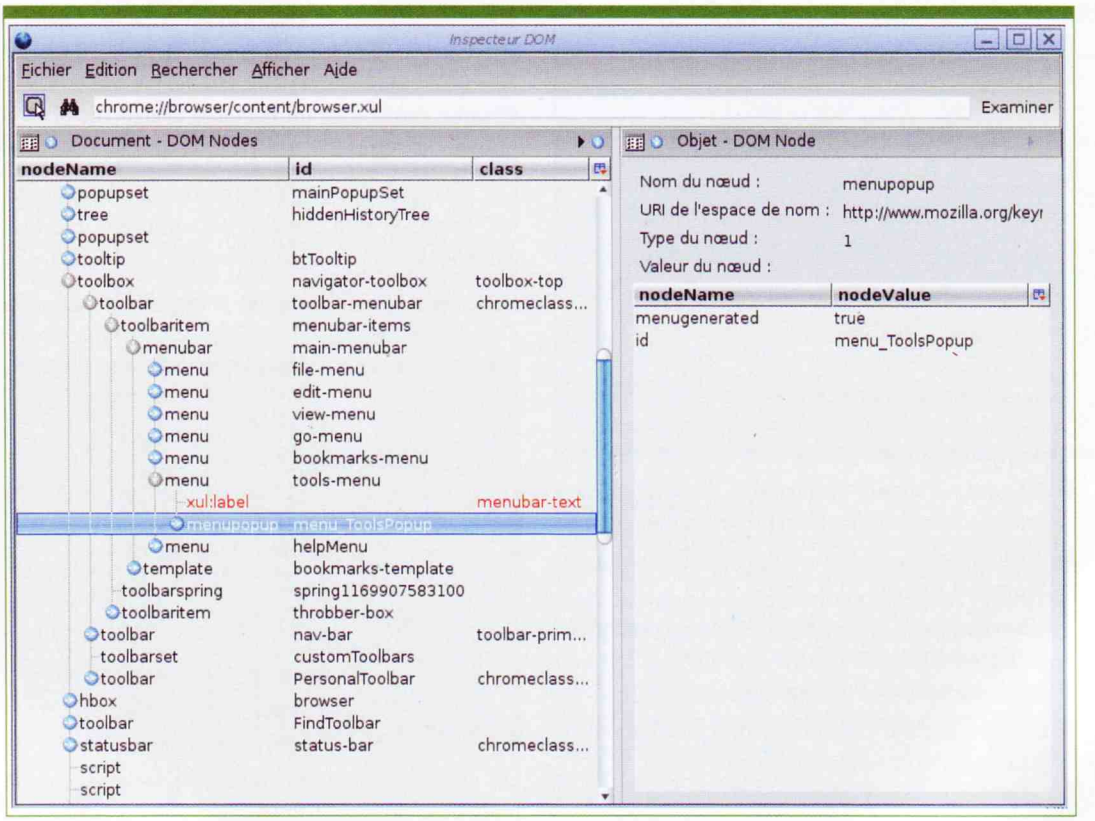
Un clic sur le bouton déclenchera l'affichage de la boîte de dialogue suivante :



Overlay

Nous allons maintenant insérer notre extension dans le menu *Outils* de Firefox. Pour ce faire, nous devons utiliser la notion d'overlay présentée dans la partie théorique de cet article. Il s'agit en fait d'une page XUL qui va étendre la page XUL principale de Firefox (**chrome://browser/content/browser.xul**). Il nous fait repérer le menu *Outils* dans la page **browser.xul**. Pour ce faire, il existe plusieurs possibilités :

- ouvrir la page dans un éditeur de texte (la page est incluse dans l'archive `$FIREFOX_INSTALL_DIR/chrome/browser.jar`) ;
- ouvrir la page dans Firefox, puis l'analyser avec l'inspecteur DOM (activé lors de l'installation avancée de Firefox et lancé via le menu *Outils/Inspecteur DOM*) ;
- ou enfin ouvrir la page dans Firefox et l'analyser avec l'extension Firebug (**http://www.getfirebug.com**).



Nous choisissons ici l'Inspecteur DOM qui nous permet de voir l'arborescence des éléments XUL de l'interface de Firefox, et de récupérer le type (`menupopup`) et l'identifiant (`menu_ToolsPopup`) du menu *Outils*. Il ne nous reste plus qu'à écrire notre page d'overlay.

Page XUL : `overlay.xul`

```
<?xml version="1.0" encoding="UTF-8"?>

<overlay id="monoverlay"
        xmlns="http://www.mozilla.org/keymaster/
gatekeeper/there.is.only.xul">

        <menupopup id="menu_ToolsPopup">
              <menuitem id="monextension_menu"
                        label="Mon extension"
                        oncommand="window.
openDialog('chrome://monextension/content/
monextension.xul', 'Mon extension', '_blank',
'chrome,dialog=no');"/>
        </menupopup>
</overlay>
```

- Nous utilisons l'élément `<overlay/>` prévu à cet effet.
- Nous indiquons que l'on veut ajouter quelque chose dans le menu *Outils*, en reprenant l'élément que l'on a trouvé via le DOM inspector : `<menupopup id="menu_ToolsPopup">`
- Dans cet élément, nous insérons un nouvel élément `<menuitem/>`.
- L'élément de menu `<menuitem/>` est ajouté avec les attributs suivants :
- `id` : identifiant unique de l'élément de menu.
- `label` : titre de l'élément tel qu'il apparaîtra dans le menu.
- `oncommand` : action à déclencher lorsque l'élément de menu est sélectionné. Ici, le code JavaScript lance l'ouverture de la page XUL de notre extension (**monextension.xul**) dans une fenêtre séparée, avec le titre `'Mon extension'`. Nous reviendrons plus loin sur l'URL passée en paramètre de la fonction `window.openDialog()`.

Notre entrée de menu sera intégrée dans le menu *Outils* de Firefox sans avoir pour autant modifié la page principale du navigateur **browser.xul**. C'est toute la puissance de l'overlay !

Packaging

Reste désormais à créer le paquet XPI en accompagnant nos deux pages XUL des fichiers suivants :

chrome.manifest

Déclaration des composants de l'extension dans le référentiel chrome de Gecko :

```
content monextension chrome/content/
overlay chrome://browser/content/browser.xul
chrome://monextension/content/overlay.xul
```

- La directive `content` enregistre l'emplacement à utiliser lors de la résolution de l'URI **chrome://monextension/content/**. Nos deux fichiers XUL doivent donc se trouver dans le répertoire `chrome/content/`. Ainsi, l'URI **chrome://monextension/content/monextension.xul**, présente dans le fichier d'overlay plus haut, pointe désormais sur la page XUL de notre extension.
- La directive `overlay` enregistre notre fichier d'overlay comme devant s'appliquer à la page principale **browser.xul** de Firefox.

install.rdf

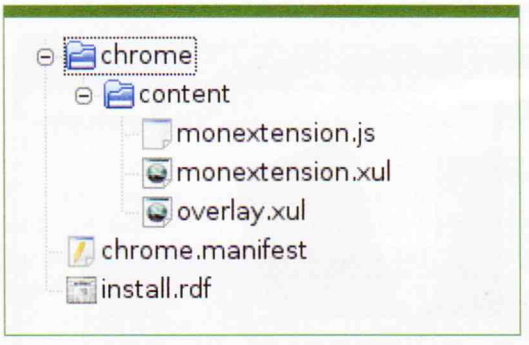
Indications d'installation à destination du gestionnaire d'extensions de Firefox :

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:em="http://www.mozilla.org/2004/em-rdf#">
       <Description about="urn:mozilla:install-manifest">
              <em:id>monextension@test.org</em:id>
              <em:version>1.0</em:version>
              <em:type>2</em:type>
              <em:targetApplication>
                     <Description>
                            <em:id>{ec8030f7-c20a-464f-9b0e-
13a3a9e97384}</em:id>
                            <em:minVersion>1.5</em:minVersion>
                            <em:maxVersion>2.0.0.*</em:maxVersion>
                     </Description>
              </em:targetApplication>
              <em:name>Mon extension</em:name>
             <em:description>Ma première extension Firefox</em:description>
       </Description>
</RDF>
```

- La balise `<em:id/>` définit de manière unique l'identifiant de l'extension. Depuis Firefox 1.5, cet identifiant peut être de la forme **nom_extension@organisation.tld**.
- La balise `<em:version/>` définit la version de l'extension.
- La balise `<em:type/>` précise, via la valeur *2*, qu'il s'agit d'une extension.
- La balise `<em:targetApplication/>` précise dans quelle application l'extension peut être installée :
  - `<em:id/>` : identifiant unique de l'application concernée (ici Firefox avec l'identifiant `{ec8030f7-c20a-464f-9b0e-13a3a9e97384}`) ;
  - `<em:minVersion/>` et `<em:maxVersion/>` : versions minimale et maximale de Firefox supportées par l'extension. Notez qu'il est recommandé que la version maximale soit celle d'une version existante, et non une version majeure future, pour éviter que l'utilisateur puisse l'installer sur des versions de Firefox pour lesquelles l'extension n'a pas été testée.
- La balise `<em:name/>` définit le nom de l'extension telle qu'elle apparaîtra dans le gestionnaire d'extensions de Firefox.
- La balise `<em:description/>` précise une courte description de l'extension pour le gestionnaire d'extensions de Firefox.

Archive XPI

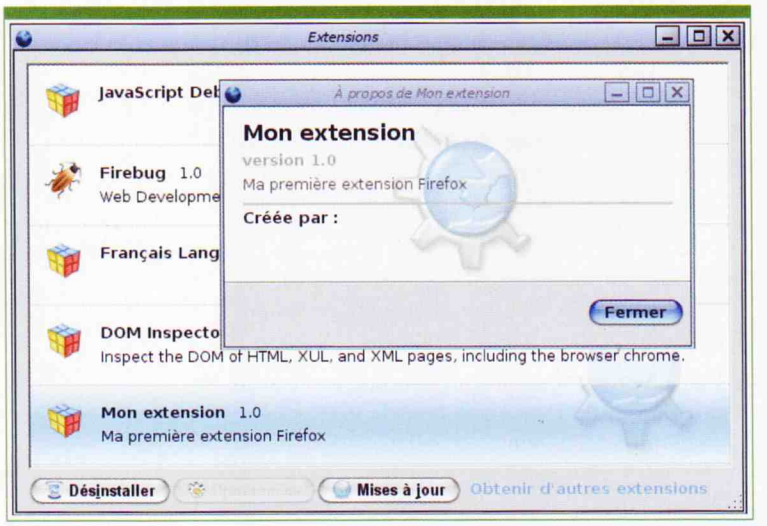
Nous avons donc la hiérarchie suivante :



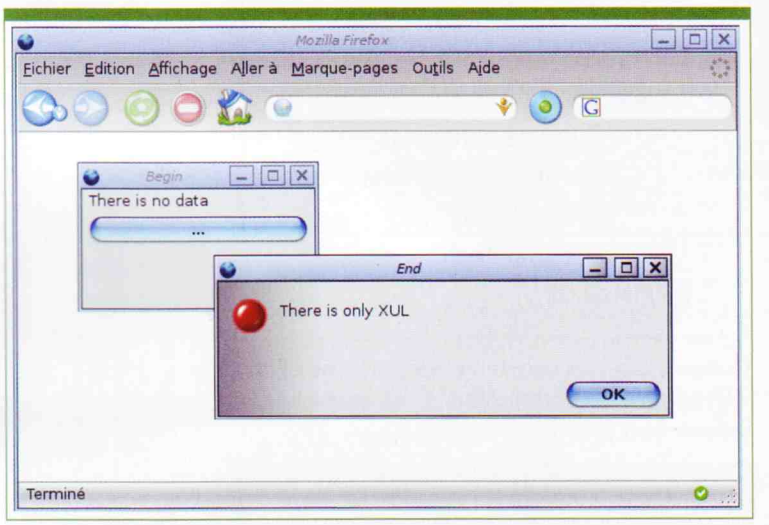
Compressons tout ce beau monde dans une archive ZIP nommée `monextension.1.0.xpi` et voilà notre extension est prête à être installée !

Ouvrons le fichier XPI, en local, à partir de Firefox, une fenêtre de confirmation s'affiche :

Une fois l'extension installée et Firefox redémarré, nous pouvons vérifier que le gestionnaire d'extensions a bien enregistré notre extension...



...et qu'elle peut être lancée depuis le menu *Outils*.



Localisation

Améliorons maintenant notre extension en y ajoutant la gestion du multi-langage. La plate-forme Mozilla propose un mécanisme spécifique pour gérer ceci : il faut séparer les chaînes de caractères des pages XUL, les informations de localisation sont alors stockées dans l'arborescence `chrome/locale` à raison d'un sous-répertoire par langue. Pour notre extension, nous implémenterons le support des localisations *fr-FR* et *en-US*.

Le fichier `chrome.manifest` doit être modifié en conséquence en ajoutant deux directives `locale` :

```
content  monextension                          chrome/
content/
overlay  chrome://browser/content/browser.xul
chrome://monextension/content/overlay.xul
locale   monextension   en-US                  chrome/
locale/en-US/
locale   monextension   fr-FR                  chrome/
locale/fr-FR/
```

Localisation de l'IHM

Les chaînes comprises dans le code XUL sont externalisées dans des fichiers DTD sous la forme `<!ENTITY key "value">` et sont référencées dans les pages XUL sous la forme `&key;`. Il faut également ajouter une ligne aux pages XUL pour y associer le fichier DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<!DOCTYPE window SYSTEM "chrome://monextension/locale/monextension.dtd">

<window id="page" title="&window.title;"
       xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
        width="200px" height="100px">

       <script type="application/x-javascript" src="monextension.js"></script>
        <vbox>
               <label value="&label.value;"/>
               <button label="..." oncommand="show();"/>
        </vbox>
</window>
```

chrome/monextension.xul

Vous remarquerez que dans l'URI chrome de la DTD, on n'indique pas la langue. Quand il s'agit d'une URI chrome de type « locale », Gecko fait automatiquement le « mapping » vers le fichier de la langue activée dans Firefox par l'utilisateur.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE overlay SYSTEM "chrome://monextension/locale/overlay.dtd">

<overlay id="monoverlay"
       xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

        <menupopup id="menu_ToolsPopup">
               <menuitem id="monextension_menu"
                         label="&menuitem.label;"
                         oncommand="window.openDialog('chrome://
monextension/content/monextension.xul', '&dialog.title;', '_blank',
'chrome,dialog=no');"/>
        </menupopup>
</overlay>
```

chrome/overlay.xul

```
<!ENTITY window.title "Début">
<!ENTITY label.value "Il n'y a pas de Data">
```

chrome/locale/fr-FR/monextension.dtd

```
<!ENTITY menuitem.label "Mon extension">
<!ENTITY dialog.title "Mon extension">
```

chrome/locale/fr-FR/overlay.dtd

```
<!ENTITY window.title "Begin">
<!ENTITY label.value "There is no Data">
```

chrome/locale/en-US/monextension.dtd

```
<!ENTITY menuitem.label "My extention">
<!ENTITY dialog.title "My extention">
```

chrome/locale/en-US/overlay.dtd

Localisation des messages JavaScript

Les messages inclus dans les scripts sont eux aussi externalisés dans un fichier `.properties` constitué d'entrées de type `key=value`. Les chaînes sont récupérées d'une manière différente, car dans un script les entités DTD ne peuvent être utilisées. Le langage XUL offre l'élément `<stringbundle/>` pour permettre à un script de récupérer et manipuler la liste des chaînes stockées dans un fichier `.properties`.

Pour ce faire, il faut :

- insérer dans le fichier XUL concerné une ligne du type `<stringbundle id="properties" src="chrome://chemin/locale/fichier.properties/">` ;
- dans le code JavaScript, récupérer l'élément `<stringbundle/>` et utiliser sa méthode `getString()` pour récupérer la valeur d'une chaîne :

```
var strbundle=document.getElementById("properties");
var value=strbundle.getString("key");
```

Dans notre cas, il nous faut donc créer un fichier `monextension.properties` par localisation.

```
title=Fin
text=Il n'y a que XUL
```

chrome/locale/fr-FR/monextension.properties

```
title=End
text=There is only XUL
```

chrome/locale/en-US/monextension.properties

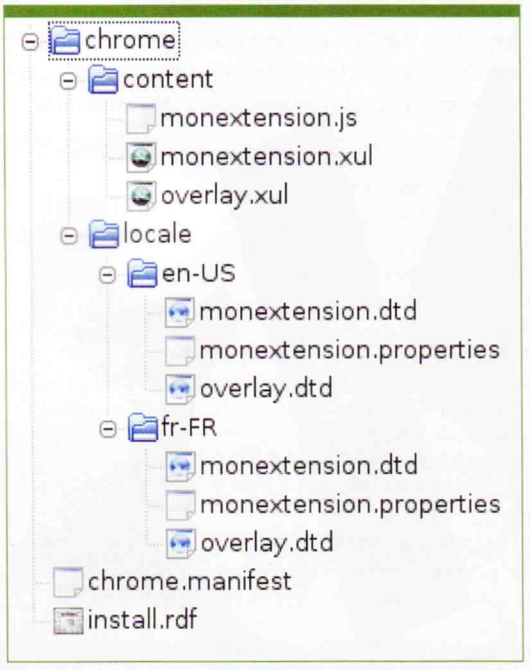
Puis insérer un élément `<stringbundle/>` dans la page `monextension.xul`.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<!DOCTYPE window SYSTEM "chrome://monextension/locale/monextension.dtd">

<window id="page" title="&window.title;"
   xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
   width="200px" height="100px">

   <script type="application/x-javascript" src="monextension.js"></script>
   <stringbundle id="properties" src="chrome://monextension/locale/
monextension.properties"/>

   <vbox>
         <label value="&label.value;"/>
         <button label="..." oncommand="show();"/>
   </vbox>
</window>
```

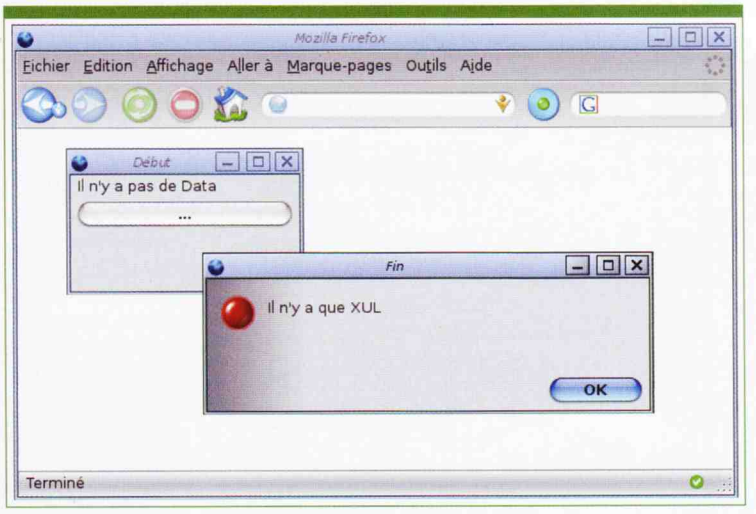
Et enfin modifier le script `monextention.js`.

```
function show() {
   var strbundle = document.getElementById("properties");
   var prompts = Components.classes["@mozilla.org/embedcomp/prompt-
service;1"]
         .getService(Components.interfaces.nsIPromptService);
   prompts.alert(null, strbundle.getString("title"), strbundle.
getString("text"));
}
```

L'arborescence de notre extension est désormais la suivante :



Il ne reste plus qu'à compresser cette arborescence dans une archive XPI, puis à réinstaller l'extension (supprimer l'ancienne version, puis installer la nouvelle depuis le gestionnaire d'extensions de Firefox) et le tour est joué : Firefox se base automatiquement sur sa localisation pour déterminer la localisation à utiliser pour l'extension.



Infrastructure de mise à jour

Nous terminerons cet article en illustrant une fonctionnalité très intéressante offerte par la plate-forme Mozilla : la gestion des mises à jour de notre extension. En effet, la manipulation manuelle réalisée plus haut dans le gestionnaire d'extensions de Firefox est fastidieuse : supprimer l'ancienne version, puis installer la nouvelle nécessite de redémarrer deux fois Firefox.

Il est possible de préciser à Firefox, via le fichier `install.rdf`, à quelle URI se connecter pour vérifier l'existence de mises à jour de notre extension. Ceci est réalisé par l'ajout d'une balise `<em:updateURL/>` contenant un lien vers un fichier manifeste indiquant les mises à jour disponibles pour cette extension.

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:em="http://www.mozilla.org/2004/em-rdf#">
        <Description about="urn:mozilla:install-manifest">
            <em:id>monextension@test.org</em:id>
            <em:version>1.0</em:version>
            <em:type>2</em:type>
            <em:targetApplication>
               <Description>
                   <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
                   <em:minVersion>1.5</em:minVersion>
                   <em:maxVersion>2.*</em:maxVersion>
               </Description>
            </em:targetApplication>
            <em:name>Mon extension</em:name>
            <em:description>Ma première extension Firefox</em:description>
            <em:updateURL>http://www.test.org/monextension/monextension-
update.rdf</em:updateURL>
        </Description>
</RDF>
```

install.rdf

Dans notre exemple, le manifeste de mises à jour sera stocké à l'adresse : http://www.test.org/monextension/monextension-update.rdf.

Ce fichier, en RDF, liste les différentes mises à jour disponibles :

```
<?xml version="1.0"?>
<r:RDF xmlns:r="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       xmlns="http://www.mozilla.org/2004/em-rdf#">
        <r:Description about="urn:mozilla:extension:monextension@test.org">
        <updates>
            <r:Seq>
            <r:li>
            <r:Description>
                <version>1.1</version>
                <targetApplication>
                    <r:Description>
                        <id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</id>
                        <minVersion>1.5</minVersion>
                        <maxVersion>2.0.0.</maxVersion>
                        <updateLink>http://www.test.org/monextension/
monextension-1.1.xpi</updateLink>
                    </r:Description>
                </targetApplication>
            </r:Description>
            </r:li>
            </r:Seq>
        </updates>
        <version>1.1</version>
        <updateLink>http://www.test.org/monextension/monextension-1.1.xpi</
updateLink>
        </r:Description>
 </r:RDF>
```

On remarquera que deux espaces de noms différents sont utilisés (`RDF` et `em-rdf`) et que les balises `<version/>` et `<updateLink/>` sont dupliquées. Ceci est nécessaire pour raison de compatibilité avec l'ancien mécanisme de mise à jour (Firefox 1.0).

Détaillons le contenu du fichier `monextension-update.rdf` :

- La première balise `<r:Description/>` précise l'extension concernée par la (ou les) mise(s) à jour. On retrouve donc dans l'attribut `about` l'identifiant unique de notre extension : `monextension@test.org`.
- Les balises `<r:Seq/>` et `<r:li/>` permettent de déclarer une liste de mises à jour. Ici nous n'en déclarons qu'une seule.
- La balise `<version/>` précise le numéro de version de la mise à jour.
- La balise `<targetApplication/>` détaille la version de Firefox pour laquelle la mise à jour est destinée (on y retrouve donc les balises `<id/>`, `<minVersion/>` et `<maxVersion/>` déjà utilisées dans le fichier d'installation `install.rdf` de notre extension.
- Enfin, la balise `<updateLink>` contient le lien vers l'archive XPI de la nouvelle version de l'extension.

On remarque donc qu'il est possible de déclarer des mises à jours pour différentes versions de l'extension et pour différentes versions de Firefox.

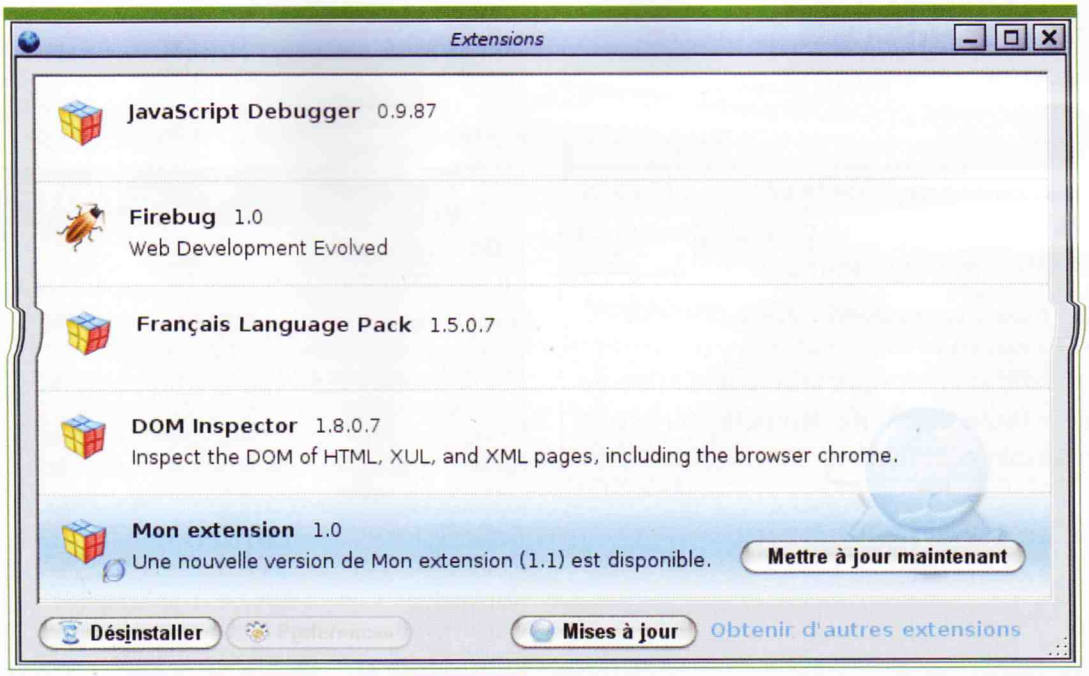
Une fois l'extension régénérée avec le nouveau fichier `install.rdf` et installée dans Firefox, il reste ensuite à déployer les fichiers `monextension-update.rdf` et nos mises à jour sur le serveur http://www.test.org.

Il est nécessaire de configurer le serveur HTTP de manière à ce qu'il associe le type MIME `text/xml` à l'extension de fichier `rdf`. Dans le cas d'Apache, ceci est réalisé via la directive :

```
AddType text/xml rdf
```

qui peut être intégrée à la configuration d'Apache ou stockée dans un fichier `.htaccess` (si vous ne pouvez modifier la configuration d'Apache de votre hébergeur par exemple).

Les mises à jour de notre extension sont désormais détectées par le gestionnaire d'extensions de Firefox !



Raphaël Semeteys,

Consultant chez Atos Origin

LIENS ET OUTILS UTILES

Voilà, notre brève introduction au développement d'extensions Firefox s'achève. Nous n'avons pas détaillé ici l'ensemble des options et des possibilités offertes par la plate-forme Mozilla. Pour plus de détails sur le développement d'extensions, je vous recommande notamment les sites suivants :

- http://xulfr.org/xulplanet/xultu/ : le tutoriel XUL traduit en français sur l'excellent site http://xulfr.org.
- http://developer.mozilla.org/fr/docs/Extensions : la section française du site *Mozilla Developper Center* réservée au développement d'extensions.
- http://www.xulplanet.com/references/xpcomref/ : la référence des composants et des interfaces XPCOM de Mozilla.

Un certain nombre d'outils sont disponibles pour faciliter le développement d'extensions. On retiendra par exemple :

- L'inspecteur DOM, à activer lors de l'installation avancée de Firefox, que nous avons vu plus haut.
- L'extension « Firebug » (http://www.getfirebug.com) qui propose de nombreuses fonctionnalités dont le débogage de code JavaScript.
- L'extension « CSView » (http://xulfr.org/outils/) qui permet de lister tous les composants et interfaces XPCOM disponibles.

Un grand merci à Laurent Jouanneau, de l'association XULFR, pour sa relecture.

Damien Krotkine

Mason – deuxième partie

Voici le deuxième article de la série sur Mason, un framework basé sur mod_perl, qui permet de créer des sites web dynamiques. Après avoir vu les bases de Mason dans l'opus précédent, nous allons aborder deux sujets importants : la Programmation Orientée Objet et la mise en place du système de cache. Nous terminerons par un exemple d'utilisation du module Ajax.

Mason orienté objet

Introduction

NOTE

La POO en Perl

Voici quelques références pour apprendre la programmation orientée objet en Perl :

- En français : l'article « La programmation objet en Perl » par Sylvain Lhullier, dans GLMF n°47, février 2003.
- En anglais : l'excellent « *Tom's object-oriented tutorial* » accessible sur toute distribution Linux qui se respecte via : `man perltoot`.

Mason implémente des fonctionnalités qui permettent de faire de la programmation orientée objet. Cette implémentation est critiquable sur le plan de sa rigueur, car tous les paradigmes ne sont pas présents. Cependant, elle est largement suffisante pour proposer l'*encapsulation*, l'*héritage* et le *polymorphisme*. Ces concepts permettent de repenser totalement le design technique d'un site web et de son implémentation.

Voici la définition de la programmation objet donnée par Wikipédia :

« La programmation orientée objet est une façon d'architecturer une application informatique en regroupant les données et les traitements de ces dernières au sein des mêmes entités, les objets. »

Le lecteur sans aucune connaissance en programmation Orientée Objet pourra se référer aux liens proposés. Une connaissance basique, que ce soit en Perl ou dans un autre langage, est suffisante pour la suite de cet article.

Le concept

Attention, ici nous parlons d'autre chose que de l'orientation objet en Perl. En effet, il y a de fortes chances pour qu'un site web Mason utilise un jeu de modules Perl *ad hoc*, qui peuvent utiliser de l'orienté objet. On peut également, dans les balises Mason, manipuler des objets Perl. Mais ce dont nous parlons ici, c'est de considérer le *composant Mason* comme un *objet à part entière*.

NOTE

Abus de langage

Il est courant d'utiliser le terme « composant » ou « objet » pour parler de la même chose : une instance d'un composant Mason ou l'objet qu'il représente. Il est également courant de parler de « classe » ou de « composant », car c'est le source du composant qui décrit la classe de l'objet. Rassurez-vous, le contexte permet généralement de lever l'ambiguïté sur le sens du mot « composant ».

Utilisons un exemple simple pour expliciter le concept. Considérons le site web d'une entreprise, qui contient deux parties. L'une d'elles est la présentation des employés, l'autre est la présentation des produits. En plus, il y a une page légale sur l'entreprise.

Un exemple

Analyse du problème

Toutes les pages de ce site web contiennent un en-tête, un bas de page, et un titre. Les pages des employés sont structurées de la même manière : le nom de l'employé, sa photo, et sa fonction. Quant aux pages produits, elles contiennent le nom du produit, sa photo et son prix.

Ainsi, on peut en déduire les classes suivantes :

- la classe `page_web` ;
  C'est la page de base, qui contient un titre, un en-tête et un bas de page.
- la classe `nom_photo` ;
  Cette page hérite de `page_web`, elle présente un nom et une photo.
- la classe `employe` ;
  Elle hérite de `nom_photo`, et en plus implémente la fonction de l'employé.
- la classe `produit`.
  Elle hérite de `nom_photo`. Elle implémente le prix.

Le titre, le nom, la photo et la fonction sont des attributs, car ils ne nécessitent pas de traitements. En revanche, l'en-tête, le bas de page et le prix seront des méthodes. On peut imaginer récupérer le prix de chaque produit dans une base de données, par exemple.

Structure des fichiers

Après avoir fait cette analyse, il est maintenant possible de lister tous les composants (donc les classes) Mason du site et d'avoir la structure des fichiers du site web :

```
/index.html      # la page d'entrée du site, hérite de page_web.html
/pageweb.html        # implémente "titre", "entete", "basdepage"
/pageweb/legal.html  # la page d'informations légale du site, hérite de pageweb
/pageweb/nom_photo.html   # hérite de pageweb, implémente "nom" et "photo"
```

```
/pageweb/nom_photo/employe.html  # hérite de nom_photo, implémente "fonction"
/pageweb/nom_photo/employe/jean.html          # page d'un employé
/pageweb/nom_photo/produit.html  # hérite de nom_photo, implémente "prix"
/pageweb/nom_photo/produit/cuillere.html      # page d'un produit
/pageweb/nom_photo/produit/couteau.html       # page d'un produit
```

`/pageweb.html`, `/pageweb/nom_photo.html`, `/pageweb/nom_photo/employe.html` et `/pageweb/nom_photo/produit.html` ne doivent pas être appelées directement. La structure des fichiers rappelle celle des modules CPAN. C'est voulu, mais ce n'est pas obligatoire. C'est une convention que nous recommandons de suivre, sous peine de se perdre facilement dans le code du site web.

Chaîne d'appel

Le principe de l'orienté objet Mason est le suivant : lors de l'appel de `/pageweb/nom_photo/produit/cuillere.html`, une chaîne d'appel est créée. Mason la détermine en partant du composant appelé, et en remontant suivant l'héritage, puis la redescend pour générer l'affichage. Lors de la remontée, Mason agrège les propriétés de l'objet avec ses parents. Une fois arrivé au sommet de la chaîne, il peut la redescendre et afficher le contenu des composants, car il a toutes les informations nécessaires. Voici la partie descendante de la chaîne :

```
pageweb.html, nom_photo.html, produit.html, cuillere.html
```

Attention, contrairement à la programmation orientée objet en Perl, ici, tous les éléments de la chaîne d'héritage seront appelés. Il n'y a pas besoin d'appeler son parent.

Par contre, un composant doit spécifiquement appeler ses éventuels enfants. En effet, les composants enfants insèrent du HTML, et il est du ressort du parent de spécifier où le HTML des enfants va apparaître, en utilisant cette syntaxe **[1]** :

```
%# on appelle le reste de la chaîne
% $m->call_next;
```

Nous allons implémenter tous les composants de ce site web, mais, d'abord, voyons quelle est la syntaxe des différents aspects de la programmation orientée objet.

Attributs, méthodes, héritage, SELF, PARENT

Les attributs et les méthodes sont des éléments indispensables à la programmation orientée objet. Il faut également avoir un moyen de spécifier l'héritage d'un objet par rapport à un autre. Voyons comment tout cela est fait dans Mason :

Attributs

Les attributs permettent d'accrocher une caractéristique simple à un objet. La syntaxe est la suivante :

```
<%attr>
nom => 'Dupuis'
prenom => 'Jean'
tableau => [1, 2, 3]
gestionnaire => sub { ... }
</%attr>
```

Il est possible de stocker n'importe quel type Perl dans un attribut, même une fonction anonyme.

Voici comment récupérer la valeur d'un attribut, dans une section `<%perl>` :

```
$composant->attr('titre')
```

D'où vient `$composant` ? Plusieurs méthodes existent pour accéder à un objet composant dans une section `<%perl>` :

- `$m->current_comp`
  Renvoie le composant courant.
- `$m->fetch_comp(chemin)`
  Renvoie le composant correspondant au chemin.
- `$m->base_comp`
  Voir plus bas.

Méthodes

Les méthodes permettent d'effectuer des tâches sur l'objet. Ici on parle de méthode Mason, et non Perl, car c'est le composant Mason qui est considéré comme objet. Il nous faut donc une méthode Mason faisant partie du composant.

```
<%method afficher_titre>
Ceci est le titre de la page
% my $foo = 2 + 2
<% $foo %>
</%method>
```

La syntaxe est simple, et permet d'encapsuler du code Mason dans une méthode, qu'il sera possible d'appeler ultérieurement. Appeler une méthode Mason est très similaire à l'appel d'un composant. Il suffit de rajouter le suffixe `:methode` :

```
<& composant:methode &>
<& /produits/cuillere:afficher_titre &>
```

Héritage

L'héritage se spécifie dans un composant par l'utilisation de la section `<%flag>` :

```
<%flags>
inherit => '../parent.html'
</%flags>
```

Cela permet de spécifier que le composant en cours hérite de `../parent.html`. Mason ne supporte pas l'héritage multiple, contrairement à Perl. Certains diront que c'est une fonctionnalité, et non une limitation :).

SELF

Il est intéressant d'avoir la main sur soi-même lorsqu'on est dans le code de l'objet. C'est ce qu'on appelle généralement *self*.

En Perl, ça sera une variable `$self`. En Mason, `SELF` est le composant lui-même (en fait son instance). On l'obtient grâce à une méthode de `$m` :

```
my $self = $m->base_comp;
```

Comment fait-on, dans un composant, pour appeler une de ses méthodes ? Mason propose un raccourci lors des appels de méthodes :

```
<& SELF:entete &>
```

Ce code appelle la méthode entete du composant courant.

PARENT

De la même manière que pour SELF, Mason propose :

```
my $parent = $m->current_comp->parent
```

pour manipuler le composant parent, et

```
<& PARENT:methode &>
```

pour appeler une méthode du parent.

Implémentation

Bien, maintenant que nous avons analysé notre exemple de site web, et que nous avons les informations pour implémenter les aspects objets de nos composants, nous pouvons commencer à les implémenter.

On commence par /pageweb.html :

```
<html>
<head>
%# Le titre est affiché à partir de l'attribut 'titre'
  <title></title>
</head>
<body>
%# l'entete, implémenté plus bas
<& SELF:entete &>

%# on appelle le reste de la chaîne (et donc les
enfants de ce composant)
% $m->call_next;

%# le bas de page
<& SELF:footer &>
</body>
</html>

<%init>
%# on a besoin de $self pour accéder aux attributs my
$self = $m->base_comp;
</%init>

<%attr>
titre => 'titre par défaut'
</%attr>

<%method header>
%# voici un header simpliste : on affiche le titre
<h2><% $self->attr('titre') %></h2>
</%method>

<%method footer>
%# un bas de page vide
</%method>
```

On peut tout de suite coder la page /pageweb/legal.html :

```
<%flags>
%# ce composant hérite de pageweb
inherit => '../pageweb.html'
</%flags>

<%attr>
titre => 'Mentions légales'
</%attr>

Corps de la page : ici on mentionne le copyright, etc.
```

On peut voir que ça permet de raccourcir la taille du code, et d'être donc plus maintenable. Passons maintenant à /pageweb/nom_photo.html :

```
<%flags>
inherit => '../pageweb.html'
</%flags>

%# on affiche le nom et l'image de la photo
<h2><% $self->attr('nom') %>
<img src='/images/<% $self->attr('image') %>'>

%# on appelle le reste de la chaîne
% $m->call_next;

<%init>
  my $self = $m->base_comp;
</%init>
```

Voyons le source de /pageweb/nom_photo/employe.html qui doit implémenter la fonction de l'employé :

```
<%flags>
inherit => '../nom_photo.html'
</%flags>

  <%attr>
titre => 'Page des employés (titre par défaut)'
</%attr>

<% $self->attr('nom') %> a pour fonction <% $self-
>attr('fonction') %>.

%# on appelle le reste de la chaîne
% $m->call_next;

<%init>
  my $self = $m->base_comp;
</%init>
```

On peut ajouter la page de l'employé Jean /pageweb/nom_photo/employe/jean.html :

```
<%flags>
inherit => '../employe.html'
</%flags>

  <%attr>
nom => 'Jean'
image => 'jean.jpg'
fonction => 'directeur général'
</%attr>

Ceci est la page de Jean.
```

Voici le code de /pageweb/nom_photo/produit.html. Le prix sera extrait d'une base de données.

```
<%flags>
inherit => '../nom_photo.html'
</%flags>

<%attr>
titre => 'Page des produits (titre par défaut)'
</%attr>

<% $self->attr('nom') %> a pour prix <& SELF:get_prix &>
%# on appelle le reste de la chaîne
% $m->call_next;
<%init>
  my $self = $m->base_comp;
</%init>

<%method get_prix>
%# par défaut on va chercher le prix dans une base
de données, et l'afficher
...
</%method>
```

On peut maintenant définir les produits, tout d'abord /pageweb/nom_photo/produit/cuillere.html

```
<%flags>
inherit => '../produits.html'
</%flags>
```

```
<%attr>
nom => 'Cuillere'
image => cuillere.jpg'
</%attr>
Ceci est une cuillère. Son prix est extrait de la base de données
```

Maintenant faisons pareil pour `/pageweb/nom_photo/produit/couteau.html`. Cependant, ici, le prix sera fixe, et non pas récupéré dans la base de données. Pour cela, on surcharge la méthode `get_prix` de `produit.html`.

```
<%flags>
inherit => '../produits.html'
</%flags>
<%attr>
nom => 'couteau'
image => 'couteau.jpg'
</%attr>
<%method get_prix>
57
</%method>
```

La programmation orientée objet de Mason est très intéressante, car elle permet d'utiliser le même mode de programmation entre le frontal et le dorsal (qui est très souvent développé en orienté objet) d'un site web. Cependant, cette méthode nécessite un petit temps d'adaptation par rapport à la programmation orientée objet de Perl.

Il est généralement de bon augure de réfléchir à la réécriture d'un site web existant en utilisant cette technique. Elle permet dans bien des cas de pointer des erreurs de conception, et de simplifier la gestion et l'amélioration future du site.

Système de cache

Théorie

La force des *frameworks* web dynamiques est de générer du contenu *à la volée*. Ainsi le site web vit de lui-même, se met à jour tout seul en fonction de différents flux de données (par exemple une base de données, ou bien une livraison électronique de nouvelles).

Cependant, cette puissance a un prix : cela coûte de la ressource matérielle, notamment du temps processeur. Et ceci peut être ennuyeux. En effet, si un serveur web n'arrive pas à générer les pages web aussi vite que les requêtes arrivent, le site peut devenir inaccessible.

Pour essayer d'éviter ce problème, on peut spécifier au framework qui fait tourner le site web, que les pages web de tel répertoire sont statiques, donc qu'elles doivent être livrées telles quelles, sans être analysées ou générées. En fait, on dit au framework d'ignorer ces pages. De cette manière, il s'occupe des pages dynamiques, et laisse le serveur web faire son travail pour les pages statiques, ce qui est bien plus rapide.

Pourtant, cette manière de faire ne résout pas vraiment le problème de ressources des serveurs web typiques, car la plupart des sites ne contiennent que très peu de pages vraiment statiques.

Depuis plusieurs années, la plupart des sites web conséquents utilisent un procédé simple à mettre en œuvre, et efficace, pour diminuer les besoins en ressources du serveur : il s'agit du système de cache. Le principe est simple : au lieu de générer une page du site pour chaque visiteur, on la crée une seule fois, et on garde le HTML généré. Ce contenu généré (ou statique) est alors utilisé pour les autres visiteurs de cette page. Ainsi, on peut allier la puissance d'un site dynamique à la performance d'un site statique.

Mettre en place un système de cache amène son lot de problèmes, dont voici une liste des plus importants :

- Mettre en place un système de cache coûte toujours un peu de temps de configuration. Même s'il est extrêmement réduit avec Mason, il peut être utile de prendre en compte ce coût.
- Un système de cache prend de la place sur le disque dur du serveur, et quelquefois augmente la consommation mémoire lors de la génération du contenu statique. Pour certains sites web où tout est mis en cache, les quantités de données stockées sur le disque dur peuvent être énormes.
- Le système de cache doit vérifier quand le contenu *statique généré doit être révoqué. Si une page* web affiche la date du jour, on ne peut la mettre en cache qu'un jour, il faudra la régénérer le jour suivant, la date ayant changé. Pour des pages très dynamiques, mettre en place un système de cache ne sera pas d'une grande utilité, et pourra complexifier inutilement la configuration du site web.

Le problème crucial est sans doute le dernier mentionné : il est important de savoir si cela vaut le coup d'activer un système de cache pour telle ou telle page du site web. Pour cela, il faut pouvoir juger du degré de dynamisme de la page web : en clair, plus une page change souvent de contenu, moins il sera rentable de la mettre en cache.

On a souvent recours à une classification erronée des pages d'un site web : d'un côté les pages dynamiques, qui nécessitent d'être générées à la volée, et, de l'autre, les pages statiques, qui ne changent jamais sur le site web. En analysant plus finement le contenu des sites web, on se rend compte qu'il faut prendre en considération sa fréquentation, et classer les pages en fonction de leur fréquence de changement par rapport à la fréquentation.

Par exemple, si en moyenne, une page A change de contenu une fois par heure, et que 50 personnes la consultent toutes les heures, on obtient le ratio *changement de contenu* par rapport à la *fréquentation* de *1/50* pour une heure, ce qui est très faible. Cette page peut être considérée comme peu dynamique, et il sera rentable de la mettre en cache.

Une autre page B, qui change tous les jours, mais qui n'obtient un taux de fréquentation que de 5 personnes par jour, aura un ratio de *1/5* pour un jour, donc de *4.8* pour une heure. Comparativement à la page A, la page B est en fait très dynamique, alors que son contenu ne change qu'une fois par jour.

Malgré les apparences, il est plus intéressant de mettre en cache la page A qui change toutes les heures, que la page B, qui change tous les jours.

Mise en œuvre

Le système de cache de Mason utilise en fait le module `Cache::Cache` de DeWitt Clinton. Plus précisément, Mason utilise un sous-module. Par défaut, ce sera `Cache::FileCache`. Ce module stocke les données à mettre en cache dans des fichiers, en les triant par nom de composant Mason. Bien entendu, il est possible de changer le type de système de cache, et d'autres options, comme expliqué un peu plus loin. Mais la configuration par défaut est un bon moyen de tester et mettre en œuvre le système de cache de Mason. Par défaut donc le module de mise en cache utilisé est `Cache::FileCache`, et le cache est nommé `cache` dans le répertoire de données de Mason.

Algorithme de mise en cache

Le principe est très simple : le système de cache permet d'associer une clef à du contenu, et d'effectuer des actions sur ce couple :

- Stocker un nouveau contenu identifié par une clef. On peut stocker n'importe quel objet Perl : une chaîne, un tableau, etc. Le contenu stocké a une durée de vie, qui peut être infinie ou conditionnée. C'est ce qu'on appelle l'expiration.
- Récupérer un contenu grâce à sa clef. On peut tester si la clef existe, et si le contenu existe.

Il est important de noter que les couples clef/contenu sont propres à chaque composant. Ainsi, on est sûr de ne pas écraser les données de cache d'un autre composant.

Avant d'écrire du code, il est préférable de réfléchir à l'algorithme de mise en cache, en se posant quelques questions :

- Pour un composant donné, où écrire le code ?
  Il ne faut pas vouloir tout mettre en cache. Comme vu dans la section précédente, il est important d'implémenter la mise en cache dans les composants qui sont le plus statique. À l'intérieur du composant, on se placera généralement au début, pour pouvoir court-circuiter la génération du contenu de la page web.
- Quelle clef choisir ?
  Il faut une clef pour y associer le contenu généré du composant. Cette clef ne peut pas être arbitraire. Elle doit correspondre à un schéma qui identifie de manière unique la manière dont le composant a été appelé.
- Quelle raison invalide le contenu ?
  L'avantage de la mise en cache est qu'elle est très performante pour les pages peu dynamiques, mais elle n'empêche pas le contenu de la page de changer. Il faut à présent caractériser ce qui fait que le contenu du composant va changer, c'est-à-dire, ce qui fait que le contenu mis en cache expire, et doit être éliminé. Cela peut être une notion purement temporelle (une page de météo mise à jour toutes les 24 heures), ou cela peut être un test faisant intervenir des éléments externes (comme une base de données).

Grâce aux réponses à ces questions, on peut alors mettre au point un algorithme de mise en cache, qui ressemblera le plus souvent à ceci :

```
clef = Generer_clef()
contenu_statique = Recuperer_du_cache(clef)
SI contenu_statique == vide ALORS
  contenu_statique = Generer_le_contenu()
  condition = Generer_condition_d_expiration
  Stocker_dans_le_cache(clef, contenu_statique, condition)
FIN
Renvoyer(contenu_statique)
```

Pas de magie noire : si un contenu statique existe, on le renvoie directement, sinon on le crée avec la bonne clef et la condition d'expiration. Cette dernière peut être temporelle (« ce contenu a une durée de vie de 5 minutes »), ou bien conditionnelle (« ce contenu n'est plus valable si une news est créée entre-temps »).

Implémentation

L'objet de cache est accessible en utilisant :

```
$m->cache
```

Cet objet implémente plusieurs méthodes, que nous présentons dans ce morceau de sources :

```
my $contenu = $m->cache->get('clef');
if (!defined($contenu)) {
    ...
    $contenu = ...;
    ...
    $m->cache->set('clef', $contenu, $condition_optionnelle);
  }
```

Ainsi, avec les méthodes `set` et `get`, on peut implémenter un système de cache fonctionnel.

Voyons maintenant un exemple réel. Imaginons un site web d'information avec une page de recherche sur les archives. Cette page web est implémentée par un composant qu'on appelle pour l'occasion `recherche.html`. Ce composant prend en argument la requête de l'utilisateur. Les brèves sont stockées dans une base de données, et une méthode `Cherche_dans_DB(requete)` permet de lancer une recherche dans la base. Cette méthode renvoie une référence sur une liste de liens vers les brèves qui correspondent. Cependant, cette recherche est assez coûteuse en ressources CPU, et serait avantageusement épaulée d'une mise en cache des requêtes les plus courantes.

Nous allons écrire le code qui met en cache la requête et la liste des réponses associées. Ainsi, si la requête de recherche `"perl"` retourne une liste (A, B, C) de brèves qui traitent de ce merveilleux langage, cette liste sera stockée dans le cache. Cependant, lorsqu'une nouvelle brève D est ajoutée dans la base de donnée du site, le contenu du cache n'est peut-être plus valide. En effet, si la brève D répond aux critères de recherche (ici `"perl"`), la requête devrait renvoyer (A, B, C, D), or la réponse mise en cache ne contient pas D.

Un moyen simple de résoudre ce problème est d'invalider

l'information mise en cache dès qu'une nouvelle est ajoutée à la base de donnée. Pour cela, on peut demander à la base de données le dernier ID des brèves, via une méthode `Recupere_dernier_ID()`. Si cet ID change, alors le contenu généré doit être invalidé. Comment implémenter cela ? Simplement en ajoutant cet ID à la clef du contenu mis en cache.

Voici un prototype du code de `recherche.html` :

```
% Composant recherche.html
% argument : la requête de recherche
<%args>
$requete
</%args>
<html>
<body>
Voici les résultats de la requête S<"<% $requete %>" :>
<%perl>
# construction de la clef, constituée de la requête,
et du dernier ID
my $dernier_id = Recupere_dernier_ID();
my $clef = $requete . '|' . $dernier_id;
# on teste s'il existe un contenu déjà en cache
my $contenu = $m->cache->get($clef);
if (!defined($contenu)) {
    $contenu = Cherche_dans_DB($requete);
    # on stocke le contenu avec un délai
d'expiration de 10 jours
    $m->cache->set($clef, $contenu, '10days');
}
# ici, on a récupéré ou généré $contenu, qui est en
fait une ref. sur liste
my @liste = @$contenu;
</%perl>
%# on peut maintenant l'afficher en HTML
<ul>
% foreach (@liste) {
    <li></li>
%}
</ul>
</body>
</html>
```

Finalement, ce n'était pas si difficile. Cependant, on notera que l'on a mis un délai d'expiration arbitraire, 10 jours. On considère en effet qu'il est improbable qu'aucune nouvelle brève n'apparaisse en 10 jours. Au bout de 10 jours, le cache sera nettoyé **[2]**. Si jamais un contenu est effacé à tort, ce n'est pas grave, le pire qui puisse arriver est qu'il soit régénéré une fois pour rien tous les dix jours.

Expiration conditionnelle

Fixer un délai d'expiration arbitraire est une solution acceptable dans beaucoup de cas, mais nous allons examiner deux méthodes pour invalider des données. Tout d'abord, au lieu d'utiliser une expiration purement temporelle (ici, 10 jours), utilisons une clause d'expiration conditionnelle, avec `expire_if`.

`expire_if` prend en argument une méthode anonyme, qui est appelée avec l'objet `cache` comme unique paramètre. Si la méthode renvoie une valeur vraie, alors le contenu est invalidé, et effacé du cache.

Ici, nous allons tester si le dernier ID de la base de données est toujours le même. Grâce à `expire_if`, nous n'avons plus besoin de stocker l'ID dans la clef. Voici la partie Perl du source précédent, modifié pour l'occasion :

```
<%perl>
# construction de la clef, constituée de la requête uniquement
my $clef = $requete
# on teste s'il existe un contenu déjà en cache
my $contenu = $m->cache->get($clef);
if (!defined($contenu)) {
    $contenu = Cherche_dans_DB($requete);
    # on récupère le dernier ID
    my $dernier_id = Recupere_dernier_ID();
    # on stocke le contenu avec un test d'expiration sur l'ID
    $m->cache->set($clef, $contenu, expire_if => sub { Recupere_
dernier_ID() != $dernier_id; });
}
# ici, on a récupéré ou généré $contenu, qui est en fait une ref. sur liste
my @liste = @$contenu;
</%perl>
```

Il faut bien comprendre que `$dernier_id` contient le dernier id *au moment de la mise en cache initiale* du contenu. Cet `id` sera ensuite comparé au dernier `id` courant lors de chaque appel de la sous-routine `expire_if`. En effet, la fonction anonyme `sub { Recupere_dernier_ID() != $dernier_id; }` est ce qu'on appelle une « fermeture ». Elle hérite du contexte dans lequel elle est créée, notamment des variables lexicales.

Invalidation externe

Il existe une dernière manière d'invalider du contenu dans le cache, en utilisant le module de cache utilisé par Mason (par défaut `Cache::FileCache`), dans un programme Perl, qui peut être à l'extérieur du site web.

Mason fournit un module utilitaire qui permet d'accéder facilement aux informations mises en cache par un composant. Il faut lui spécifier le nom du composant, et l'endroit où est stocké le cache de Mason (par défaut, c'est le sous-répertoire *cache* de `MasonDataDir`, spécifié dans le fichier de configuration d'Apache **[3]**).

```
# On charge les fonctions relatives au cache du module utilitaire
use HTML::Mason::Utils qw(data_cache_namespace);
# création d'un nouvel objet Cache
my $cache = new Cache::FileCache( {
        namespace  => data_cache_namespace('/chemin/du/composant'),
        cache_root => '/var/cache/apache/cache'
} );
# Effacer une clef spécifique
$cache->remove('clef1');
# Effacer tout le cache pour ce module
$cache->clear();
```

Les possibilités offertes par l'accès externe au système de cache sont multiples. Ainsi, une tâche de nettoyage du cache peut être déclenchée à intervalle régulier (configuré dans la `crontab` du système par exemple), sans avoir besoin d'interagir avec `mod_perl`. On peut également déclencher l'effacement d'une clef

du cache, lorsque la donnée associée est effacée de la base de donnée.

Configuration du système de cache

Par défaut, Mason utilise `Cache::FileCache` et les données du cache sont stockées dans `data_dir/cache`. Ces paramètres (et d'autres) peuvent être changés dans le fichier de configuration d'Apache, via `MasonDataCacheDefaults` :

```
PerlSetVar MasonDataCacheDefaults "cache_class => MemoryCache"
PerlAddVar MasonDataCacheDefaults "cache_depth => 2"
PerlAddVar MasonDataCacheDefaults "default_expires_in => 1 hour"
```

- `cache_class` permet de changer la classe utilisée pour la mise en cache. `File::Cache`, `MemoryCache` sont des exemples. `NullCache` peut être utilisé, cela permet de désactiver le cache totalement, sans changer une ligne de code, très utile pour trouver un bug, sans que la mise en cache interfère.
- `cache_depth` permet de préciser la profondeur maximale des composants qui vont pouvoir utiliser le système de cache.
- `default_expires_in` permet de préciser le délai d'expiration par défaut.

Ajax et Mason

Introduction

Sur le site principal de Mason, on peut trouver des composants à télécharger et utiliser sur nos sites web. Il s'agit d'exemples de code, ou de composants génériques permettant de répondre à des problématiques fréquentes. L'un d'eux traite d'Ajax. Il est accessible ici : **http://www.masonhq.com/?Component:ajax**.

Il permet d'utiliser le principe d'Ajax sans écrire de JavaScript ou alors très peu. Ce composant (appelé `ajax`) utilise la bibliothèque JavaScript `Prototype` pour gérer l'appel asynchrone, et il permet d'implémenter la réponse en Perl, et non en JavaScript. Voyons un exemple qui permet de détailler le fonctionnement de ce composant.

Tout d'abord, il nous faut télécharger ce module depuis **http://www.masonhq.com/?Component:ajax**, et le sauvegarder dans un composant Mason, par exemple à la racine du site web, **/var/www/localhost/htdocs/mason.html**. Ainsi, il sera accessible à l'adresse **http://www.siteweb.com/mason.html**. Il faut également se procurer la bibliothèque JavaScript `Prototype` depuis **http://www.prototypejs.org/** et l'installer dans un répertoire du site web, par exemple **/var/www/localhost/htdocs/js/prototype-1.4.0.js**.

Pour illustrer l'utilisation de ces technologies, nous allons construire une page web qui permet d'afficher une image au hasard.

Récupérer une image aléatoire

Pour cela, il nous faut tout d'abord une banque d'images accessible sur le net, avec un lien permettant d'en afficher une au hasard. Vous connaissez sûrement le site web de l'excellent Ayo (**http://www.73lab.com/**), qui produit de très jolies images, fonds d'écran et autres éléments graphiques. Son site web met à notre disposition une adresse spéciale, qui permet d'afficher une vignette de fond d'écran au hasard : **http://ayo73.free.fr/random_pic/random_pic.php3** **[4]**. Voici un exemple du contenu HTML renvoyé par cette page :

```
<a href="http://www.73lab.com" border=0 target=new>
  <img
    src="http://ayo73.free.fr/random_pic/pics/05_ayodacode_epreuve_
1280_1024_thumb.jpg"
    border=0 width=120 height=90
  >
</a>
```

Il suffit de récupérer l'attribut `src` du tag `img`, et nous obtenons l'adresse d'une image prise au hasard. Voici le code que nous pouvons utiliser pour faire ce travail :

```
use LWP::Simple;   # utilisation de modules perl
use URI;           # pour faciliter le travail
# on charge l'URL du site d'Ayo, et on ne prend que l'attribut src du tag img
my ($url_image) = (get("http://ayo73.free.fr/random_pic/random_
pic.php3") =~ /img src="(.*?)"/);
# on extrait le nom de l'image de son adresse
my $nom_image = (URI->new($url_image)->path_segments())[-1];
# enfin, on télécharge l'image, et on la place dans un repertoire accessible
# en écriture par Apache.
mirror($url_image, "/var/www/localhost/htdocs/images/$nom_image");
```

Ce code devra être placé dans une méthode, pour pouvoir être appelé facilement.

Page d'affichage HTML

Concernant notre page d'exemple à proprement parler, côté HTML, nous allons faire simple : un `div`, et un bouton, qui, lorsqu'il est cliqué, va télécharger une image au hasard et changer le contenu du `div` pour l'afficher. L'image va être téléchargée sur le serveur web, et une balise `img` pointant sur sa copie locale va être insérée dans la balise `div`. Voici le HTML de la page :

```
<html>
<head>
  <title>Test d'Ajax avec HTML::Mason</title>
</head>
<body>
  <h1>Test d'Ajax avec HTML::Mason</h1>
<input type="button" value="image aléatoire">
<br/>
<br/>
<div id="div_image">cliquez !</div>
<br>
```

Associer les deux

À présent, il nous faut lier les deux parties, le HTML et le code qui récupère l'image. Lors d'un clic sur le bouton, il faut télécharger une nouvelle image, la stocker localement, et ajouter un tag `<img>` dans le `<div>`, qui est caractérisé par son identifiant `"div_image"`.

Pour cela, quelques modifications sont à faire. Nous allons tout d'abord ajouter un événement sur le clic du bouton, qui effectue un appel au composant `ajax.html`. Cet appel spécifiera la méthode Mason à utiliser

pour créer du HTML, et que ce HTML de retour doit être injecté dans le tag `"div_image"`. Voici comment modifier le tag `input` :

```
  <input type="button" value="image aléatoire"
   onClick="<& /ajax.html, comp=>'SELF:telecharge_
image_aleatoire', update=>'div_image' &>;";>
```

L'option `comp` spécifie le composant et la méthode qu'il faut appeler lors du clic. Ici, nous indiquons que la méthode est `telecharge_image_aleatoire`, qui est dans le composant courant.

L'option `update` permet de spécifier dans quel tag le HTML de retour doit être injecté. Nous utilisons `div_image`.

Pour que le JavaScript généré par le composant `ajax` fonctionne correctement, il faut que la bibliothèque `Prototype` soit chargée. Il faut donc également ajouter une ligne qui inclut cette bibliothèque.

Nous pouvons à présent écrire l'intégralité de notre page web de test, en incluant le code Perl de récupération d'une image au hasard. Voici le source final :

Le code final

NOTE

L'option `throbber`

Elle permet d'afficher une image animée pendant le rafraîchissement. Cette image GIF doit être placée par défaut à la racine du site web, et s'appeler `throbber.gif`. Lors de l'appel Ajax, l'élément du DOM sera remplacé par ce GIF animé pendant le temps de chargement.

```
  %# code de la page test_ajax.html

  %# méthode qui incorpore le code Perl vu précédemment
  <%method telecharge_image_aleatoire>

  <%perl>
    use LWP::Simple;
    use URI;
    my ($url_image) = (get("http://ayo73.free.fr/random_pic/random_
pic.php3") =~ /img src="(.*?)"/);
    my $nom_image = (URI->new($img_url)->path_segments())[-1];
    mirror($img_url, "/var/www/localhost/htdocs/images/$nom_image");
  </%perl>

  %# ici nous retournons le code HTML qui affiche l'image locale
  <img src=''/>

  </%method>
  <html>
  <head>
    <title>Test d'Ajax avec HTML::Mason</title>
   <script src="/var/www/localhost/htdocs/js/prototype-1.4.0.js"></script>
  </head>

  <body>
    <h1>Test d'Ajax avec HTML::Mason</h1>
  <input type="button" value="image aléatoire" onClick="
  <& /ajax.html, comp=>'SELF:telecharge_image_aleatoire', throbber
=> 1, update=>'div_image' &>;
  ";>
  <br/>
  <br/>
  <div id="div_image">Cliquez !</div>
  <br>
```

Voici une capture d'écran **[5]** du résultat :

Utilisation d'Ajax en Mason

Testing Ajax in HTML::Mason

random image

Le module `ajax` utilisé présente plusieurs avantages. Tout d'abord, il permet de modifier le contenu de la page courante sans la recharger. Mais en plus, il permet de le faire avec pratiquement pas de JavaScript. Enfin, la syntaxe Mason permet de rassembler le code à exécuter sur le serveur et l'affichage du contenu dans un même composant, ce qui le rend plus simple à écrire et à maintenir.

Conclusion

Nous avons examiné quelques fonctionnalités avancées de Mason dans cet article, qui en font un framework complet. Mason dispose également d'un certain nombre de modules d'extensions, touchant à des sujets variés, disponibles sur le CPAN, dans le groupe `MasonX`. Dans le prochain article, nous verrons quelques-uns de ces modules utilisés dans un cas concret.

Damien Krotkine,

<dams@zarb.org> – Paris.pm

LIENS

- MasonHQ : http://www.masonhq.com
- Le composant Ajax : http://www.masonhq.com/?Component:ajax
- La bibliothèque Prototype : http://www.prototypejs.org
- Le site d'Ayo : http://www.73lab.com/

NOTES

[1] Voir l'article précédent pour une description de `$m`.

[2] Attention, il ne s'agit pas du cache complet qui sera nettoyé, mais seulement de l'élément du cache concernant la requête dont on parle. Ce nettoyage à granularité fine permet d'éviter au serveur web d'effectuer une grande quantité d'accès disque au même moment, comme cela aurait été le cas si l'ensemble du cache avait été invalidé au bout de 10 jours.

[3] Voir, dans l'article précédent, la partie sur la configuration d'Apache.

[4] On pardonnera à Ayo d'utiliser du PHP, cette tare étant largement compensée par ses talents de graphiste.

[5] Image utilisée avec l'aimable autorisation d'Ayo.

Yves Bailly

Le langage Ada – 18 : systèmes distribués

Voilà un titre qui évoque sans doute Corba, DCop et autres DBus ou encore Mosix ou Beowulf. Toutes ces technologies reposent sur un ensemble de bibliothèques, en général en C/C++. Pourtant, depuis sa deuxième version en 1995, la norme du langage Ada prévoit diverses facilités pour construire un programme distribué en n'utilisant que des fonctionnalités du langage.

Dans le texte du standard Ada, il s'agit de « l'Annexe E », aussi appelée « *Distributed Systems Annex* ». Cette annexe définit une sémantique et un ensemble de constructions permettant la mise en œuvre d'un programme réparti sur différentes *partitions*, chacune de ces partitions pouvant se trouver sur une machine physique différente – ou pas. Toutefois, toutes les implémentations d'Ada ne supportent pas les fonctionnalités définies dans l'Annexe E. C'est heureusement le cas du compilateur de référence utilisé pour cette série d'article, le compilateur GNAT, en lui adjoignant un complément, GLADE.

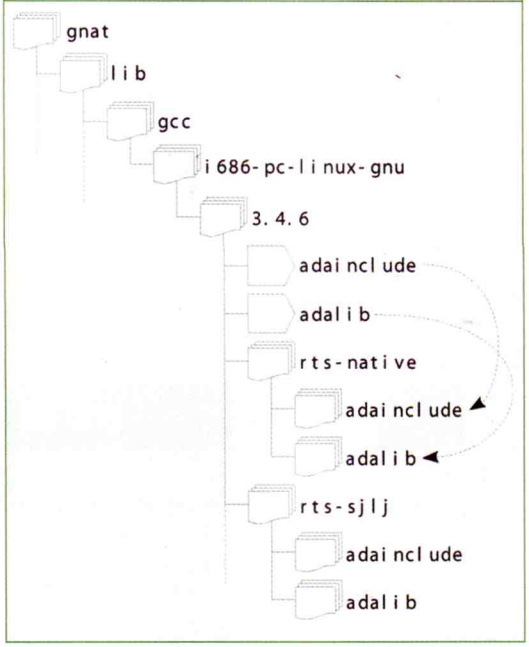
Il est intéressant de noter que la norme du langage ne définit réellement qu'un ensemble de comportements, pas la manière de les réaliser effectivement. Cet aspect est donc laissé à la discrétion de chaque compilateur – ou plutôt de chaque chaîne de compilation. Ainsi, le support de la programmation distribuée ne fait-il pas partie intégrante de GNAT, mais est plutôt fourni sous la forme d'un paquetage à part, nommé GLADE (*GNAT Library for Ada Distributed Execution*). La manière de compiler les programmes change un peu, comme on pouvait s'y attendre. Mais en dehors de cette spécificité, les aspects du langage que nous allons voir aujourd'hui sont généraux et utilisables sur tout compilateur supportant l'Annexe E.

Installation de GLADE

Pour commencer, si ce n'est déjà fait, récupérez la dernière version du compilateur GNAT telle que distribuée par AdaCore **[1]**. Au moment où ces lignes sont écrites, il s'agit de la version de juillet 2006. Sur la même page de téléchargement, vous trouverez un lien vers les sources de GLADE : `glade-gpl-2006-src.tgz`. Téléchargez également ce fichier.

Installez GNAT à l'emplacement de votre choix, par exemple `$HOME/gnat`. Pour compiler et utiliser GLADE, il est nécessaire de réaliser maintenant une petite adaptation. Placez-vous dans le répertoire `$HOME/gnat/lib/gcc/i686-pc-linux-gnu/3.4.6`. Normalement, vous y trouverez deux liens symboliques, `adalib` et `adainclude`, pointant vers des sous-répertoires de même nom de `rts-native` :

Figure 1 : Traitement d'une interruption



L'adaptation consiste à déplacer ces liens vers les sous-répertoires dans `rts-sjlj` :

```
$ rm adainclude adalib
$ ln -s rts-sjlj/adainclude/ adainclude
$ ln -s rts-sjlj/adalib/ adalib
```

Ce faisant, on change les caractéristiques du *runtime* GNAT utilisé, en particulier le modèle utilisé pour les exceptions. Le modèle par défaut est « *zero-cost* », mais le bon fonctionnement de GLADE impose l'utilisation du modèle « *setjmp/longjmp* » (merci Xavier pour l'information !). Le détail de chaque implémentation sort du cadre de cet article ; disons simplement que le modèle zero-cost est le plus rapide et le plus efficace, tandis que le modèle setjmp/longjmp est probablement le plus robuste et le plus dynamique.

Cela fait, décompactez l'archive des sources de GLADE, puis placez-vous dans le répertoire nouvellement créé :

```
$ tar zxf glade-gpl-2006-src.tar.gz
$ cd glade-2006-src
```

Si ce n'est pas déjà le cas, ajoutez le répertoire contenant les binaires de GNAT à votre `PATH` :

```
$ export PATH=$HOME/gnat/bin:$PATH
```

Puis configurez et installez GLADE dans le même répertoire que GNAT :

```
$ ./configure --prefix=$HOME/gnat
$ make && make install
```

Vous pouvez également installer GLADE dans un autre répertoire. Pensez seulement à ajouter le sous-répertoire `bin` dans votre `PATH`.

Si tout se passe bien, vous pouvez maintenant réaliser des programmes distribués.

Premier programme (local)

Commençons par le commencement, c'est-à-dire le traditionnel programme saluant le monde. Celui-ci va afficher la chaîne « Bonjour, Monde » en concaténant les chaînes renvoyées par deux fonctions, chacune contenue dans un paquetage différent.

Voici le paquetage `Pkg_Bonjour`, qui fournit la première partie de la chaîne :

```
package Pkg_Bonjour is
   function Bonjour
      return String;
end Pkg_Bonjour;
```

Et son implémentation :

```
package body Pkg_Bonjour is
   function Bonjour
      return String is
   begin
      return "Bonjour";
   end Bonjour;
end Pkg_Bonjour;
```

Difficile de faire plus trivial. Nous avons un paquetage `Pkg_Monde`, tout à fait équivalent. Le programme principal utilise les deux paquetages ainsi :

```
with Ada.Text_IO; use Ada.Text_IO;
with Pkg_Bonjour;
with Pkg_Monde;
procedure BM is
begin
   Put_Line(Pkg_Bonjour.Bonjour &
            ", " &
            Pkg_Monde.Monde &
           " !");
end BM;
```

Compilez ce programme, exécutez-le. Vous obtiendrez le résultat attendu.

Maintenant, transformons ce programme « monobloc » en programme distribué. Les modifications nécessaires dans le code source sont d'une indicible complexité : il suffit d'ajouter la ligne

```
pragma Remote_Call_Interface
```

...au début des spécifications des paquetages. Celles-ci deviennent alors :

```
package Pkg_Bonjour is
   pragma Remote_Call_Interface;
   -- etc.
```

Et de même pour le paquetage `Pkg_Monde`. Cette directive signale que le contenu de ce paquetage peut être invoqué à distance. Notez que vous pouvez toujours compiler « normalement » votre programme, pour l'instant cela ne change rien.

La distribution du programme est paramétrée par un fichier de configuration. Celui-ci décrit comment les différents paquetages vont être répartis entre les différentes partitions, ainsi que les nœuds où seront situées ces partitions. Voici un exemple de configuration pour notre programme, placé dans le fichier `bonjour_monde.cfg` :

```
configuration Bonjour_Monde is
   pragma Starter(Ada);
   dist_bm: Partition := ();
   procedure BM is in dist_bm;
   bonjour: Partition := (Pkg_Bonjour);
   for bonjour'Host use "localhost";
   monde: Partition := (Pkg_Monde);
   for monde'Host use "localhost";
end Bonjour_Monde;
```

La première partition déclarée est celle contenant notre procédure principale, la procédure `BM` (normalement dans le fichier `bm.adb`). Cette partition est déclarée vide (par `:= ()`), car elle ne contient en fait aucun paquetage, aucun sous-programme appelable à distance. Il est toutefois nécessaire d'indiquer qu'elle contient la procédure principale, ce qui est fait par `procedure BM is in dist_bm`.

La seconde partition, nommée `bonjour`, correspondra au paquetage `Pkg_Bonjour`. La syntaxe permet de donner plusieurs paquetages dans les parenthèses : une même partition peut ainsi « contenir » plusieurs paquetages. Par contre, un même paquetage de type `Remote_Call_Interface` ne peut être associé qu'à une seule partition. La ligne `for bonjour'Host use...` indique la machine qui sera chargée d'exécuter le code de cette partition – pour l'instant, la machine locale sera bien suffisante, ne brûlons pas les étapes.

Enfin, la troisième partition correspond au paquetage `Pkg_Monde`, sur le même modèle que la précédente. Créons maintenant notre programme distribué à l'aide de l'utilitaire `gnatdist`. Vous devriez obtenir quelque chose comme ceci :

```
$ gnatdist bonjour_monde.cfg
gnatdist: checking configuration consistency
 ----------------------------
 ---- Configuration report ----
 ----------------------------
Configuration :
   Name        : bonjour_monde
   Main        : bm
   Starter     : Ada code

Partition dist_bm
   Main        : bm
   Units       :
               - bm (normal)

Partition bonjour
   Host        : localhost
   Units       :
               - pkg_bonjour (rci)

Partition monde
   Host        : localhost
   Units       :
```

```
            - pkg_monde (rci)

 ----------------------------
gnatdist: building pkg_bonjour caller stubs from
pkg_bonjour.ads
gnatdist: building pkg_bonjour receiver stubs from
pkg_bonjour.adb
gnatdist: building pkg_monde caller stubs from pkg_
monde.ads
gnatdist: building pkg_monde receiver stubs from
pkg_monde.adb
gnatdist: building partition dist_bm
gnatdist: building partition bonjour
gnatdist: building partition monde
gnatdist: generating starter bm
```

Des informations sont données pour chaque partition et les unités qu'elles contiennent. Ainsi, la première partition `dist_bm` (correspondant à la procédure principale) est qualifiée de `"normal"`, tandis que les unités des deux autres partitions sont qualifiées par `"rci"`, pour *Remote Call Interface* (interface d'appel distant). À l'issue de cette phase de construction, vous devez avoir dans le répertoire courant quatre exécutables : `bm` (le « *starter* »), `dist_bm` (première partition), `bonjour` (deuxième partition) et `monde` (troisième partition), ainsi qu'un sous-répertoire `dsa` (pour *Distributed Systems Annex*) qu'il vous est recommandé de ne pas toucher. Ce répertoire contient des fichiers intermédiaires créés lors de la compilation. Lorsque la mise au point est terminée, vous pouvez naturellement l'effacer.

Le « starter » est le programme responsable du démarrage les autres partitions – ici, il se note `bm`, comme notre programme principal. Démarrez-le :

```
$ ./bm
Bonjour, Monde !
```

Rien d'extraordinaire direz-vous... Mais si on examine les processus en cours d'exécution (ajoutez une ligne `delay 60.0;` à la fin de la procédure principale) :

```
$ ps -U yves f -o pid,args
  PID COMMAND
...
19086 /home/yves/.../bonjour --detach --boot_location
tcp://10.0.0.100:55738
19084 /home/yves/.../monde --detach --boot_location
tcp://10.0.0.100:55738
...
14478 kdeinit Running...
14483  \_ klauncher [kdeinit] -new-startup
...
17994  \_ kate [kdeinit]
17996      \_ /bin/bash
19082          \_ ./bm
...
```

On retrouve bien nos trois partitions (les chemins ont été ici abrégés), nommément la partition principale `bm` et les partitions `bonjour` et `monde`. Nous avons donc bien un programme en trois morceaux. Mais peut-être des esprits chagrins estimeront-ils que c'est un peu facile, car tout se passe sur la même machine...

Distribution

Nous allons supposer ici disposer de trois hôtes (trois machines) : la machine principale, de nom `kafka`, ainsi que deux autres systèmes de noms `bonjour-host` et `monde-host`, ces trois machines communiquant correctement sur un réseau (l'utilisation des adresses IP n'est pas suffisante, il est nécessaire que les noms soient correctement configurés). Pour les essais qui vont suivre, la machine principale dispose d'une distribution Debian Etch avec (presque) tous les outils de développement et bureautique possibles, tandis que les deux autres machines résultent d'une installation fraîche d'une distribution Ubuntu Server 6.10 minimaliste. Autrement dit, elles ne disposent de guère plus que d'un *shell* et d'un serveur OpenSSH, point de serveur X et encore moins d'outils de développement. Sur chacune de ces deux machines un compte utilisateur `yves` a été créé, auquel il est possible de se connecter par `ssh`. Une configuration réseau mauvaise ou incomplète est la source principale des difficultés que vous pourriez rencontrer dans vos expérimentations.

Pour commencer, changeons les adresses des partitions `bonjour` et `monde` dans le fichier de configuration :

```
for bonjour'Host use "bonjour-host";
for monde'Host   use "monde-host";
```

...puis recompilons le tout avec `gnatdist`.

Il convient ensuite de placer les exécutables des diverses partitions sur les machines concernées, par exemple en utilisant `scp` (l'utilitaire de copie de fichier de OpenSSH) :

```
$ scp bonjour yves@bonjour-host:/~
$ scp monde yves@monde-host:/~
```

Pour l'instant, les exécutables sont placés dans le répertoire de l'utilisateur. Nous verrons plus tard comment les placer à un endroit quelconque.

Selon votre configuration, il est possible que cela ne suffise pas. Les exécutables dépendent de quelques bibliothèques spécifiques du compilateur GNAT et du module GLADE. Pour identifier lesquelles, utilisez `ldd` à partir de la machine principale où vous effectuez les compilations :

```
$ ldd bonjour
        linux-gate.so.1 =>  (0xffffe000)
        libgarlic.so.2006 => /usr/lib/libgarlic.so.2006 (0xa7ea9000)
        libgnarl-4.1.so.1 => /usr/lib/libgnarl-4.1.so.1 (0xa7e7b000)
        libgnat-4.1.so.1 => /usr/lib/libgnat-4.1.so.1 (0xa7c1d000)
        libpthread.so.0 => /lib/tls/libpthread.so.0 (0xa7c0b000)
        libgcc_s.so.1 => /lib/libgcc_s.so.1 (0xa7c00000)
        libc.so.6 => /lib/tls/libc.so.6 (0xa7ace000)
        libz.so.1 => /usr/lib/libz.so.1 (0xa7aba000)
        libm.so.6 => /lib/tls/libm.so.6 (0xa7a95000)
        /lib/ld-linux.so.2 (0xa7f7b000)
```

Normalement, les autres fichiers sont déjà présents sur les hôtes, sauf éventuellement les fichiers des bibliothèques Garlic, Gnarl et Gnat.

- GARLIC (acronyme de *Generic Ada Reusable Library for Interpartition Communication*) est la bibliothèque d'exécution (*runtime*) permettant le dialogue entre partitions d'un programme Ada distribué ; elle fait partie de GLADE.
- GNARL (pour *GNu Ada Runtime Library*) est la bibliothèque utilisée par GNAT pour la mise en œuvre des tâches Ada (voir *Linux Magazine* 87).
- GNAT est tout simplement le runtime du compilateur GNAT.

Si vous ne disposez pas de ces fichiers sur les hôtes devant faire fonctionner les différentes partitions, il est nécessaire de les copier manuellement, soit dans un répertoire commun comme `/usr/lib`, soit dans un répertoire dûment référencé par la variable d'environnement `LD_LIBRARY_PATH`. Ici, soucieux de ne pas (trop) perturber les systèmes des hôtes distants, nous allons les placer dans le répertoire de l'utilisateur sur chacun :

```
$ scp /usr/lib/libgarlic.so.2006 \
   /usr/lib/libgnat-4.1.so.1     \
   /usr/lib/libgnarl-4.1.so.1    \
   yves@bonjour-host:~
```

Et de même pour `monde-host`.

Mais il est également possible que vous n'ayez aucun besoin de tout cela. Cela dépend en fait de la version et de la configuration de GNAT et de GLADE.

Nous allons maintenant lancer chacune des partitions manuellement. Sur la machine principale, exécutez :

```
[kafka]$ ./bm --nolaunch --boot_location tcp://
kafka:32111
```

Le paramètre `--nolaunch` indique à `bm` qu'il ne doit pas tenter de démarrer lui-même les autres partitions, car nous allons le faire nous-même. `--boot_location` sert à identifier la localisation de la partition principale, en donnant le protocole (ici TCP), l'hôte (ici `kafka`) et le port à utiliser (ici `32111`, valeur parfaitement arbitraire, choisissez simplement un port inutilisé).

Cela fait, ouvrez une session en tant qu'utilisateur `yves` (ou celui que vous aurez choisi) sur chacune des deux autres machines. Première étape, comme nous avons préalablement copié les fichiers bibliothèques nécessaires dans le répertoire de l'utilisateur, il peut s'avérer nécessaire de renseigner la variable d'environnement `LD_LIBRARY_PATH` afin que les exécutables des partitions trouvent ces fichiers :

```
[bonjour-host]$ export LD_LIBRARY_PATH=.
[monde-host]$ export LD_LIBRARY_PATH=.
```

Enfin, lancez les partitions en leur donnant la localisation de la partition principale :

```
[bonjour-host]$ ./bonjour --boot_location tcp://kafka:32111
[monde-host]$ ./monde --boot_location tcp://kafka:32111
```

Normalement, sur la machine principale (`kafka`) le message attendu devrait s'afficher, puis les trois exécutables devraient se terminer. Et voilà, nous avons un programme distribué !

Configuration d'une automatisation

Dans une situation réelle, lancer chacune des partitions « à la main » peut s'avérer fastidieux, voire impossible. En fait, la partition principale (`bm` pour nous) est capable de démarrer elle-même les partitions distantes. Pour cela, elle se connecte sur les hôtes distants au moyen de `rsh`, l'ancêtre non sécurisé de `ssh`. Comme la sécurité est importante, il n'est pas question d'utiliser `rsh` - d'ailleurs, il ne devrait même pas être installé. Pour utiliser un autre moyen de connexion, il suffit d'ajouter une directive `Remote_Shell` dans le fichier de configuration (`bonjour_monde.cfg` dans notre exemple) :

```
pragma Remote_Shell(Command => "ssh",
                    Options => "-C");
```

On indique par là que la commande à utiliser est `ssh`, en lui passant l'option `-C` (qui active la compression des transferts). Reconstruisez le programme avec `gnatdist`, puis essayez de lancer la partition principale sans autres paramètres :

```
[kafka]$ gnatdist bonjour_monde.cfg
[kafka]$ ./bm
yves@monde-host's password:
```

Naturellement, comme si vous tentiez de vous connecter « normalement » sur chacun des hôtes, le mot de passe vous est demandé. Malheureusement, même si vous donnez les deux mots de passe (un par hôte distant), rien ne semble se passer. La raison est tout simplement que les exécutables ne sont pas trouvés : ils sont par défaut supposés se trouver dans le même répertoire sur les hôtes que sur la machine principale, où nous effectuons les compilations. Par exemple, si vous compilez dans le répertoire `/home/yves/un/sous/rep`, alors les exécutables des partitions sont supposés se trouver dans le répertoire `/home/yves/un/sous/rep` sur chacun des hôtes. Or, nous les copions dans le répertoire de l'utilisateur.

Deux solutions s'offrent à vous. Vous pouvez reproduire sur les hôtes les répertoires nécessaires et y placer les exécutables. Vous pouvez également préciser l'emplacement de ceux-ci dans le fichier de configuration :

```
for bonjour'Directory use "/home/yves";
for monde'Directory use "/home/yves";
```

Recréez le programme avec `gnatdist` : les exécutables ne sont plus construits dans le répertoire courant, mais dans le répertoire indiqué. Recopiez-les sur les hôtes distants, par exemple avec :

```
[kafka]$ scp /home/yves/bonjour yves@bonjour-host:~
```

Puis relancez le programme principal :

```
[kafka]$ ./bm
yves@monde-host's password:
yves@bonjour-host's password:
Bonjour, Monde !
```

Si vous doutez encore que l'exécution se déroule effectivement sur les machines distantes, supprimez complètement les fichiers exécutables sur la machine principale :

```
[kafka]$ rm -Rf bonjour monde dsa *.o *.ali
```

Enfin, notez qu'il est possible de configurer `ssh` de façon à ne plus devoir saisir les mots de passe à chaque exécution, par l'intermédiaire de `ssh-agent`. Voici un exemple de configuration permettant cela, à adapter selon votre situation.

Tout d'abord, il faut générer une clef d'identification sur la machine principale à l'aide de `ssh-keygen` :

```
[kafka]$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/yves/.ssh/id_dsa):
Created directory '/home/yves/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/yves/.ssh/id_dsa.
Your public key has been saved in /home/yves/.ssh/id_dsa.pub.
The key fingerprint is:
89:a5:90:0d:44:d3:fd:89:6c:18:33:d3:5c:67:50:43 yves@kafka
```

La clef publique se trouve dans le fichier `~/.ssh/id_dsa.pub`. Copions-le sur les machines distantes :

```
[kafka]$ scp /home/yves/.ssh/id_dsa.pub yves@bonjour-host:~
[kafka]$ scp /home/yves/.ssh/id_dsa.pub yves@monde-host:~
```

Il est ensuite nécessaire de placer son contenu dans le fichier `~/.ssh/authorized_keys` sur chacune de ces machines. Typiquement :

```
[bonjour-host]$ mkdir .ssh
[bonjour-host]$ cat id_dsa.pub >> .ssh/authorized_keys
[bonjour-host]$ rm -f id_dsa.pub
```

Et de même pour l'autre machine.

Enfin, il reste à démarrer `ssh-agent` sur la machine principale :

```
[kafka]$ ssh-agent
```

Puis à lui donner nos clefs utilisables, ce qui peut se faire simplement par `ssh-add` :

```
[kafka]$ ssh-add
```

Maintenant l'exécution du programme distribué devrait se faire sans aucune intervention :

```
[kafka]$ ./bm
Bonjour, Monde !
```

La technique présentée ici est assez élémentaire. Consultez les documentations d'OpenSSH et des outils afférents si vous êtes dans une situation différente.

Objets distribués

La qualification par `Remote_Call_Interface` d'un paquetage permet de faire déjà pas mal de choses, mais elle souffre de deux inconvénients majeurs : d'abord un tel paquetage ne peut exister que sur une seule partition, ensuite le partitionnement du programme est très « statique ».

La technique des objets distribués apporte une réponse à ces deux contraintes. Le principe général est le suivant :

1. déclarez un type abstrait dans un paquetage qualifié par `Pure` ou `Remote_Types` : ce type de paquetages ne contient aucun code lié à son élaboration, c'est-à-dire que lorsque le paquetage est effectivement créé lors du démarrage du programme, il n'exécute aucun code ni ne déclare de variable interne ;

2. dérivez ce type en autant de types concrets que vous avez besoin, dans autant de paquetages qualifiés par `Remote_Types` ; chacun de ces paquetages devra en outre contenir une instance du type concret ;

3. créez un paquetage de procédures distantes (`Remote_Call_Interface`) qui pourra être utilisé par les paquetages des types concrets pour « enregistrer » (*register*) leur instance.

Non, ce n'est pas si compliqué que cela en a l'air. Voyons un exemple simple. Déclarons un type abstrait contenant une simple donnée, un nombre entier :

```
package Abs_RTypes is
   pragma Pure;
   type Abs_RType is abstract tagged limited private;
   procedure Set_Data(rt: access Abs_RType'Class;
                      d : in     Integer);
   function Get_Data(rt: access Abs_RType'Class)
      return Integer;
private
   type Abs_RType is abstract tagged limited
   record
      data: Integer := 0;
   end record;
end Abs_RTypes;
```

Tout cela est très trivial. Remarquez la deuxième ligne, la qualification par `pragma Pure` du paquetage. La raison de cette qualification sera donnée bientôt. Le corps de ce paquetage ne présente rien de particulier, aussi ne le détaillerons-nous pas.

Voyons maintenant le paquetage chargé de référencer les instances des types concrets dérivant de `Abs_RType` :

```
with Abs_RTypes;
package Rci is
   pragma Remote_Call_Interface(Rci);
   type Abs_RType_Ptr is
      access all Abs_RType.Abs_RType'Class;
   procedure Add_Instance(i: in Abs_RType_Ptr);
   function Nb_Instances
      return Natural;
   function Get_Instance(ind: in Positive)
      return Abs_RType_Ptr;
end Rci;
```

Les sous-programmes déclarés ici pourront être utilisés à distance, comme le montre la directive `pragma` en troisième ligne. `Add_Instance()` permet de mémoriser (un pointeur sur) une instance d'un type dérivant de `Abs_RType`, `Nb_Instances` donne le nombre d'instances ainsi mémorisées, enfin, `Get_Instance()` retourne le pointeur préalablement mémorisé. Voici le corps de ce paquetage :

```
package body Rci is
   type Array_Instances is
      array(1..5) of Abs_RType_Ptr;
   instances: Array_Instances;
```

```
   nb_insts : Natural := 0;
   --
   procedure Add_Instance(i: in Abs_RType_Ptr) is
   begin
      nb_insts := nb_insts + 1;
      instances(nb_insts) := i;
   end Add_Instance;
   --
   function Nb_Instances
      return Natural is
   begin
      return nb_insts;
   end Nb_Instances;
   --
   function Get_Instance(ind: in Positive)
      return Abs_RType_Ptr is
   begin
      return instances(ind);
   end Get_Instance;
end Rci;
```

Comme vous pouvez le constater, il n'y a rien de vraiment sorcier là-dedans. Les pointeurs sur les instances sont stockés dans un tableau `instances` de type `Array_Instances`. Ici, nous utilisons un tableau fixe de cinq éléments, par simplicité, mais rien n'interdit d'utiliser une structure plus dynamique comme un conteneur `Vector` ou une liste chaînée (voir *Linux Magazine* 89 pour les conteneurs en Ada 2005).

Petite remarque, la notion de « pointeur » utilisée ici peut être assez éloignée de celle du langage C. En effet, nous manipulons des « pointeurs » sur des objets se trouvant sur une ou plusieurs machines distantes : cela n'a donc théoriquement *rien à voir* avec une adresse mémoire.

Nantis de ces deux paquetages, nous pouvons dès maintenant écrire un petit programme de test :

```
with Ada.Text_IO; use Ada.Text_IO;
with Abs_RTypes;
with Rci;
procedure Main is
   inst_ptr: Rci.Abs_RType_Ptr;
   data    : Integer;
begin
   delay 5.0;
   Put_Line(
      Natural'Image(Rci.Nb_Instances) &
      " instances.");
   if Rci.Nb_Instances > 0
   then
      for inst in 1..Rci.Nb_Instances
      loop
         inst_ptr := Rci.Get_Instance(inst);
         Abs_RTypes.Set_Data(inst_ptr,
                             10*inst);
      end loop;
      for inst in 1..Rci.Nb_Instances
      loop
         inst_ptr := Rci.Get_Instance(inst);
         Put("Instance " &
            Integer'Image(inst) & " ");
         data := Abs_RTypes.Get_Data(inst_Ptr);
         Put_Line(" data = " &
            Integer'Image(data));
      end loop;
   end if;
end Main;
```

Ce programme se contente d'attendre que les instances s'enregistrent par l'instruction `delay`, puis, pour chacune d'elles, une donnée est stockée, puis récupérée. Nous n'utilisons que les facilités offertes par le paquetage « contrôleur » `Rci` et le type abstrait. L'attente initiale est imposée par le temps nécessaire au démarrage effectif des différentes partitions.

Créons maintenant un type concret, dont les instances s'enregistreront auprès de `Rci` :

```
with Abs_RTypes;
package RTypes is
   pragma Remote_Types(RTypes);
   pragma Elaborate_Body;
   type RType is
      new Abs_RTypes.Abs_RType
      with private;
private
   type RType is
      new Abs_RTypes.Abs_RType
      with null record;
end RTypes;
```

Cette fois, le paquetage est qualifié par `Remote_Types`, signalant que les types qui s'y trouvent déclarés seront accessibles à distance. La clause `pragma Elaborate_Body` est nécessaire, car cette spécification ne contient aucune déclaration de sous-programme : dans ce cas, le corps du paquetage n'est normalement pas pris en compte (c'est même une erreur d'en fournir un). `Elaborate_Body` permet de « forcer » l'utilisation du corps, que voici :

```
with Rci;
package body RTypes is
   instance: aliased RType;
begin
   Rci.Add_Instance(instance'Access);
end RTypes;
```

Non, vous ne rêvez pas : cela est suffisant, dans la mesure où notre type ne surcharge ni n'ajoute d'opérations au type de base. La véritable subtilité réside dans la déclaration d'une instance de ce type, puis en l'appel à une procédure (distante) de `Rci` dans le corps du paquetage. Ainsi, lorsque le paquetage sera effectivement construit au lancement du programme, cette instance sera connue du paquetage contrôleur – et donc utilisable par le programme principal.

Il est temps de passer à la configuration :

```
configuration Config is
   pragma Starter(Ada);
   pmain: Partition := (Rci);
   p1   : Partition := (RTypes);
   p2   : Partition := (RTypes);
   p3   : Partition := (RTypes);
   for p1'Host use "localhost";
   for p2'Host use "bonjour-host";
   for p3'Host use "monde-host";
   for p2'Directory use "/home/yves";
   for p3'Directory use "/home/yves";
   procedure Main is in pmain;
   pragma Remote_Shell(Command => "ssh",
                       Options => "-C");
end Config;
```

Le paquetage `Rci` sera contenu dans la partition principale, aux cotés de la procédure principale `Main()` (ce n'est pas obligatoire, il pourrait se trouver dans une autre partition). Puis, trois autres partitions sont déclarées, chacune contenant une instance du paquetage `RTypes`, celui contenant notre type concret. Cela signifie que nous aurons *trois* instances du paquetage, donc trois instances du type `RType` – alors qu'il n'en est fait aucune mention dans le programme principal. Si tout est correctement configuré, compilez le programme, placez les exécutables `p2` et `p3` correspondant aux partitions éponymes dans les répertoires indiqués des machines distantes indiquées, puis lancez l'exécution :

```
[kafka]$ gnatdist config.cfg
...
[kafka]$ ./main
 3 instances.
Instance  1  data =  10
Instance  2  data =  20
Instance  3  data =  30
```

Nous avons bien le résultat attendu ! C'est un peu comme si nous avions lancé trois tâches distinctes, trois instances d'un même type de tâche.

Distribution dynamique

L'automatisation utilisée précédemment a un prix : il est nécessaire de connaître à l'avance le nombre de partitions nécessaires, ainsi que leurs localisations. Il est pourtant possible de lever cette contrainte, ce coté trop « statique » de ce que nous avons vu jusqu'ici.

Pour commencer, modifions légèrement notre programme principal pour créer une attente :

```
-- [...]
begin
   Put("Appuyez sur Entrée...");
   declare
      s: String := Get_Line;
   begin
      null;
   end;
   Put_Line(
      Natural'Image(Rci.Nb_Instances) &
      " instances.");
-- etc.
```

Le début de la procédure principale attend simplement l'appui sur la touche [Entrée]. Modifions ensuite ainsi la configuration :

```
configuration Config is
   pragma Starter(None);
   pmain: Partition := (Rci);
   part : Partition := (RTypes);
   procedure Main is in pmain;
end Config;
```

Nous spécifions cette fois que les différentes partitions seront démarrées manuellement, en donnant `None` à la directive `pragma Starter`. Ensuite, en plus de la partition principale, une seule partition est déclarée correspondant à notre type distribué. Compilez le tout avec `gnatdist`.

Les différents exécutables à lancer sont, cette fois-ci, nommés comme les partitions correspondantes. Copiez donc l'exécutable `part` sur les deux machines distantes que nous utilisons depuis le début de cet article. Puis, sur la machine principale, lancez la première partition en spécifiant le port à utiliser pour les communications :

```
[kafka]$ ./pmain --boot_location tcp://kafka:32111
Appuyez sur Entrée...
```

Ensuite, sur chacune des machines distantes, lancez la partition `part` avec le même paramètre :

```
[bonjour-host]$ ./part --boot_location tcp://kafka:32111
[monde-host]$ ./part --boot_location tcp://kafka:32111
```

Vous pouvez en lancer d'autres si vous le souhaitez (mais pas plus de cinq !). Enfin, appuyez sur la touche [Entrée] sur la machine principale pour stopper l'attente :

```
[kafka]$ ./pmain --boot_location tcp://kafka:32111
Appuyez sur Entrée...
 2 instances.
Instance  1  data =  10
Instance  2  data =  20
```

Nous avons donc bel et bien un objet distribué qu'il est possible de dupliquer et dont il est possible de tenir compte dynamiquement. Les deux inconvénients évoqués plus haut concernant les paquetages `Remote_Call_Interface` possèdent donc une réponse.

Exécution asynchrone

Jusqu'ici, nos programmes étaient assez linéaires dans leur exécution. Mais, il serait fort intéressant de pouvoir lancer un calcul long sur une partition, puis de le laisser se dérouler pendant que le programme fait autre chose. Considérez cet exemple minimaliste. D'abord, la spécification d'un paquetage :

```
package Asynch is
   pragma Remote_Call_Interface(Asynch);
   procedure Long_Calcul;
end Asynch;
```

Puis, son corps :

```
package body Asynch is
   procedure Long_Calcul is
   begin
      delay 5.0;
   end Long_Calcul;
end Asynch;
```

Ici, notre procédure `Long_Calcul` se contente d'attendre cinq secondes, mais imaginez une partie de la simulation d'un modèle climatique pour l'hémisphère nord.

Le programme principal ne va pas faire grand-chose de plus, si ce n'est mesurer le temps qui passe :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
with Asynch;
procedure Main is
   debut: Time;
   fin  : Time;
```

```
   duree: Duration;
begin
   Put_Line("Début...");
   debut := Clock;
   Asynch.Long_Calcul;
   delay 6.0;
   fin := Clock;
   duree := fin - debut;
   Put_Line("Temps : " &
            Duration'Image(duree));
end Main;
```

On invoque donc le `Long_Calcul` dans `Asynch`, puis on attend bêtement six secondes – mais imaginez un modèle climatique simulé pour l'hémisphère sud. Enfin, on affiche le temps écoulé. La configuration est désormais connue :

```
configuration Config is
   pragma Starter(Ada);
   pmain  : Partition := ();
   pasynch: Partition := (Asynch);
   for pasynch'Host use "bonjour-host";
   procedure Main is in pmain;
   pragma Remote_Shell(Command => "ssh",
                       Options => "-C");
   for pasynch'Directory use "/home/yves";
end Config;
```

Compilez avec `gnatdist`, placez l'exécutable `pasynch` dans le répertoire indiqué sur `bonjour-host` et exécutez :

```
[kafka]$ gnatdist config.cfg
...
[kafka]$ scp /home/yves/pasynch yves@bonjour-host:~
[kafka]$ ./main
Début...
Temps :  11.017278000
```

Ce qui est finalement le résultat attendu : cinq secondes dans `Long_Calcul`, six secondes d'attente ensuite, cela donne environ onze secondes.

Maintenant, ajoutez la ligne suivante juste après la déclaration de `Long_Calcul` :

```
   pragma Asynchronous(Long_Calcul);
```

Recompilez, recopiez, re-exécutez :

```
[kafka]$ ./main
Début...
Temps :  6.001814000
```

Le temps passé n'est plus que celui de l'attente dans le programme principal : l'exécution du corps de la procédure `Long_Calcul` a été effectuée de manière *asynchrone*, c'est-à-dire que l'appel de la procédure est (presque) instantané.

Plutôt qu'une procédure, il est possible de déclarer un type (distribué) complet comme étant asynchrone. Dès lors, ses opérations primitives polymorphes deviennent asynchrones. Cela ouvre la possibilité d'effectuer, par exemple, des calculs en parallèle, comme nous l'avons vu avec les tâches – sauf que, cette fois, on répartit le calcul sur plusieurs *machines* plutôt que sur plusieurs processeurs. Notez que chaque partition, chaque *nœud* de calcul peuvent eux-mêmes utiliser des tâches...

La seule contrainte est que les sous-programmes asynchrones ne peuvent être que des procédures dont les éventuels paramètres sont tous en mode `in`.

Stockage distribué

Dans une situation où vous utilisez la programmation distribuée pour effectuer en parallèle un gros calcul, il peut s'avérer intéressant de stocker les résultats intermédiaires en un emplacement spécialisé : certains nœuds de calcul y placeront des données, d'autres iront les récupérer pour leur usage.

Reprenons notre exemple de distribution dynamique. Ajoutons un paquetage, qui sera une partition destinée au stockage de données :

```
package Stockage is
   pragma Shared_Passive(Stockage);
   type Array_Ints is array(1..5) of Integer;
   protected Stock is
      procedure Set(pos: in Positive;
                    val: in Integer);
      function Get(pos: in Positive)
         return Integer;
   private
      ints: Array_Ints := (others => 0);
   end Stock;
end Stockage;
```

Ce paquetage est qualifié par `Shared_Passive`, qui indique que la partition correspondante sera parfaitement passive : elle n'émettra pas de requêtes aux autres partitions, elle ne pourra que répondre aux requêtes des autres partitions.

Le moyen le plus sûr pour partager des données est d'utiliser un objet protégé (`protected`), que nous avons déjà rencontré dans le cadre des tâches. Celui déclaré ici ne fait rien de plus que contenir un tableau de cinq entiers, en fournissant les sous-programmes d'accès indispensables. Le corps de ce paquetage est tout à fait trivial, aussi le passerons-nous sous silence.

Modifions légèrement le paquetage `Abs_RTypes`, qui contient la déclaration du type abstrait distribué. D'abord la spécification :

```
package Abs_RTypes is
   pragma Remote_Types(Abs_RTypes);
   type Abs_RType is abstract tagged limited private;
-- etc.
end Abs_RTypes;
```

Puis, le corps :

```
with Stockage;
package body Abs_RTypes is
   procedure Set_Data(rt: access Abs_RType'Class;
                      d : in     Integer) is
   begin
      rt.data := d;
      Stockage.Stock.Set(d, 10*d);
   end Set_Data;
   function Get_Data(rt: access Abs_RType'Class)
      return Integer is
   begin
      return Stockage.Stock.Get(rt.data);
   end Get_Data;
end Abs_RTypes;
```

Vous pouvez constater que la qualification du paquetage est passée de `Pure` à `Remote_Types`. Ces qualifications imposent diverses contraintes aux contenus des paquetages. Entre autres, un paquetage déclaré `Pure` ne peut dépendre (par `with`) que de paquetages eux-mêmes déclarés `Pure`. Or, nous voulons avoir accès à notre paquetage de `Stockage` : d'où la requalification.

Les modifications consistent simplement à placer une donnée dans le stockage en fonction de celle reçue par l'instance du type distribué. C'est cette même donnée qui sera renvoyée par la suite.

Enfin, demandons au programme principal d'afficher directement le contenu stocké, en plus de la demande faite au type distribué :

```
with Ada.Text_IO; use Ada.Text_IO;
with Abs_RTypes;
with Stockage;
with Rci;
procedure Main is
-- etc.
      for inst in 1..Rci.Nb_Instances
      loop
         inst_ptr := Rci.Get_Instance(inst);
         Put("Instance " &
            Integer'Image(inst) & " ");
         data := Abs_RTypes.Get_Data(inst_Ptr);
         stock := Stockage.Stock.Get(inst);
         Put_Line(" data = " &
            Integer'Image(data) & " (" &
            Integer'Image(stock) & ")");
      end loop;
-- etc.
```

Les deux valeurs affichées doivent normalement être égales.

Ajoutons donc une partition pour notre stockage à la configuration :

```
configuration Config is
   pragma Starter(None);
   pmain: Partition := (Rci);
   part : Partition := (RTypes);
   stock: Partition := (Stockage);
   procedure Main is in pmain;
   for stock'Allow_Light_PCS use False;
end Config;
```

Dans certaines situations, GLADE optimise le code de communication placé dans une partition. C'est notamment le cas pour les partitions purement passives, comme la partition `stock` déclarée ici (si cette partition contenait un autre paquetage « actif », comme `RTypes` ou `Rci`, cette optimisation n'aurait pas lieu). Ici, cela présente un inconvénient : comme nous allons lancer les partitions manuellement (`pragma Start(None)`), la partition de stockage se terminerait immédiatement, du fait de sa passivité optimisée. Affecter la valeur `False` à l'attribut `'Allow_Light_PCS` permet d'interdire l'optimisation.

Compilez tout cela, placez une copie de `part` sur la machine `bonjour-host` et une copie de `stock` sur `monde-host`. Lancez la partition principale :

```
[kafka]$ ./pmain -boot_location tcp://kafka:32111
```

...et rien ne se passe : la partition principale attend que la partition de stockage soit disponible, car elle en a besoin pour fonctionner. En effet, une clause `with Stockage` apparaît dans le programme principal – alors qu'aucune clause `with RTypes` n'apparaît nulle part. C'est pourquoi cette partition n'est pas considérée comme indispensable au démarrage du programme. Lancez donc la partition de stockage :

```
[monde-host]$ ./stock --boot_location tcp://kafka:32111
```

Ce n'est qu'après cela qu'apparaît enfin le message d'attente. Lancez quelques instances de la partition contenant le type distribué :

```
[bonjour-host]$ ./part --boot_location tcp://kafka:32111 &
```

...puis appuyez enfin sur [Entrée] :

```
[kafka]$ ./pmain --boot_location tcp://kafka:32111
Appuyez sur Entrée...
 3 instances.
Instance  1  data =  10 ( 10)
Instance  2  data =  20 ( 20)
Instance  3  data =  30 ( 30)
```

On a bien le résultat attendu.

Cette technique permet de partager assez facilement des données entre partitions d'un programme distribué. Dès lors, il devient possible de répartir efficacement une charge de traitement sur plusieurs machines et d'en récupérer les résultats en un point central.

Conclusion

Voilà pour cette petite présentation des systèmes distribués en Ada, selon les outils accompagnant le compilateur GNAT. Manquant de matériel, votre serviteur n'a pu effectuer de tests de performances (les « machines distantes » utilisées étaient en fait des systèmes virtualisés sur une unique machine physique). Tous les éléments sont disponibles pour créer une véritable grappe de calcul sans trop de difficultés, avec toutefois peut-être un peu moins de souplesse que les techniques plus connues pour cela.

Naturellement, on pourrait continuer longtemps sur le sujet. Notamment, nous n'avons pas abordé le contenu du paquetage `System.RPC` qui donne accès au sous-système de communication entre les partitions, ni les questions de priorité. Mais vous devriez déjà avoir de quoi expérimenter pas mal d'idées avec ce que nous avons vu : imaginez ce que vous pouvez réaliser en combinant la distribution avec les tâches Ada et le polymorphisme de la programmation objet !

RÉFÉRENCE

▶ **[1]** GNAT et GLADE sur AdaCore : https://libre.adacore.com/

Yves Bailly,

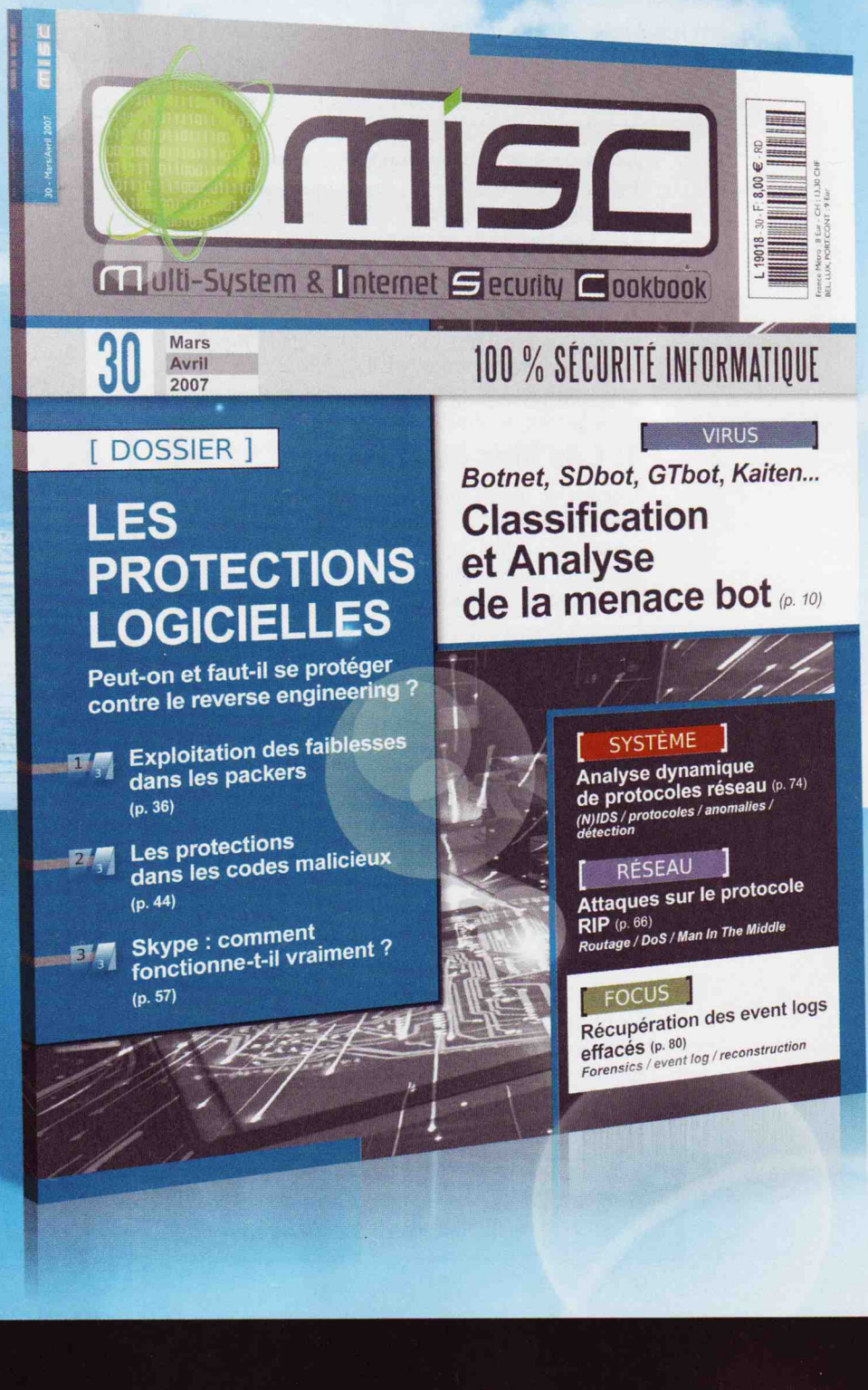
http://www.kafka-fr.net

MISC

Multi-System & Internet Security Cookbook

En KIOSQUE

NUMÉRO 30



Pierre Ficheux

Programmation noyau sous Linux
Partie 3 : techniques avancées

Ce troisième article de la série vous permettra d'aborder les sujets non traités dans l'article précédent. Certains sujets ne sont pas spécifiques aux pilotes en mode caractère, mais ils constituent des techniques fréquemment utilisées pour le développement de ces derniers qui, répétons-le, sont les plus fréquents. La lecture de cet article nécessite la connaissance des concepts décrits dans les deux précédents articles de la série.

Les exemples utilisés sont disponibles sur http://pficheux.free.fr/articles/lmf/kernel_programming/adv_drivers/exemples/tgz.

- les files de travail (ou *work queue*) ;
- la gestion des interruptions ;
- les *threads* du noyau ;
- l'accès au matériel et l'utilisation de la méthode `mmap()` ;
- la synchronisation par sémaphore et *mutex ;*

1. Les files de travail

Une file de travail (ou work queue), dans la version originale, permet d'exécuter une tâche de manière asynchrone et/ou régulière. Dans le cas de l'espace utilisateur, une telle tâche peut facilement être exécutée au travers du démon *cron* piloté par la commande `crontab`. Bien sûr, il est toujours envisageable d'utiliser cette méthode pour charger un module noyau à intervalle régulier, mais c'est loin d'être élégant.

L'API du noyau Linux fournit un composant beaucoup mieux adapté : on crée une file de type `workqueue_struct` pouvant être liée à l'exécution d'une tâche de type `work_struct` soit immédiatement, soit avec un délai (spécifié en milli-secondes). Dans le cas présent, nous exécuterons la fonction avec un délai, à la manière d'un compteur. Dans la partie concernant les interruptions, nous utiliserons une tâche exécutée « immédiatement » (sans délai), placée dans la file d'exécution.

L'exemple ci-dessous est tiré du *Linux Kernel Module Programming Guide* disponible sur http://www.tldp.org. Le principe est de créer une tâche exécutée toutes les 100 ms qui met à jour une entrée dans le système de fichier virtuel `/proc`.

Dans un premier temps, il est nécessaire de déclarer la file et la tâche associée.

```
#include <linux/workqueue.h>   /* We schedule tasks here */
/* ... */
#define PROC_ENTRY_FILENAME "sched"
#define MY_WORK_QUEUE_NAME "WQsched.c"
/* ... */
/*
 * The work queue structure for this task, from workqueue.h
 */
static struct workqueue_struct *my_workqueue;
/* Task is associated to function intprt_routine
static struct work_struct Task;
static DECLARE_WORK (Task,  intrpt_routine, NULL);
```

Dans la fonction d'initialisation du module, on crée la file et la tâche, exécutée toutes les 100 ms. On place la tâche dans la file de travail grâce à la fonction `queue_delayed_work()`.

```
/*
 * Put the task in the work_timer task queue, so it will be executed at
 * next timer interrupt
 */
my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
queue_delayed_work(my_workqueue, &Task, 100);
```

Dans la fonction associée à la tâche, on doit de nouveau armer le compteur.

```
static void intrpt_routine(void *irrelevant)
{
        /*
         * Increment the counter
         */
        TimerIntrpt++;
        /*
         * If cleanup wants us to die
         */
        if (die == 0)
                queue_delayed_work(my_workqueue, &Task, 100);
}
```

Après chargement du module par `insmod`, on peut constater l'exécution au travers du fichier virtuel `/proc/sched`.

```
# insmod sched.ko
# cat /proc/sched
Timer called 18 times so far
# cat /proc/sched
Timer called 21 times so far
# cat /proc/sched
Timer called 24 times so far
# cat /proc/sched
Timer called 26 times so far
# rmmod sched
```

2. Le traitement des interruptions

La notion d'interruption ou IRQ (pour *Interrupt ReQuest*) est très fréquemment utilisée pour la gestion des périphériques d'un calculateur. Il existe, en effet, deux manières d'interagir avec un périphérique.

1. Vérifier son état de manière régulière. Cela peut être fait au travers d'un registre d'état. Cette méthode est appelée « *polling* ». Elle est simple à mettre en œuvre, mais n'est pas envisageable dans le cas de la nécessité d'une interaction rapide ou d'un grand nombre de périphériques à traiter, car elle introduit de grandes valeurs de latence (temps de réponse du système).

2. Gérer des interruptions. Dans ce cas, le périphérique signale un changement d'état en actionnant un signal matériel dédié (une ligne d'interruption). Au temps de la préhistoire de l'informatique, chaque niveau d'interruption (une valeur numérique) était associé à un périphérique donné. De nos jours, les calculateurs sont capables de gérer le partage des interruptions, ce qui permet d'utiliser un même niveau pour plusieurs périphériques. Au niveau logiciel, les problèmes de latence sont également à considérer, car la prise en compte de l'interruption n'est bien évidemment pas immédiate.

Vu de l'espace utilisateur, les interruptions gérées par le système (donc associées à un pilote gérant l'interruption) sont visibles par le fichier virtuel `/proc/interrupts`. L'exemple ci-dessous correspond à un système de type x86 IA32.

```
# cat /proc/interrupts
           CPU0
  0:    2832085   IO-APIC-edge  timer
  1:      22977   IO-APIC-edge  i8042
  7:          1   IO-APIC-edge  parport0
  8:          1   IO-APIC-edge  rtc
  9:      76862   IO-APIC-level  acpi
 12:      16740   IO-APIC-edge  i8042
 14:     172171   IO-APIC-edge  ide0
169:     806070   IO-APIC-level  uhci_hcd:usb4, HDA Intel, nvidia
177:          0   IO-APIC-level  yenta
201:          9   IO-APIC-level  uhci_hcd:usb3, ohci1394
209:     449351   IO-APIC-level  uhci_hcd:usb1, ehci_hcd:usb5
217:          0   IO-APIC-level  uhci_hcd:usb2, sdhci:slot0
225:      27678   IO-APIC-level  skge
233:     128309   IO-APIC-level  ipw2200
NMI:          0
LOC:     917144
ERR:          0
MIS:          0
```

En cas d'interruption partagée, la liste des périphériques sera signalée sur la colonne de droite, les éléments étant séparés par des virgules. La première colonne correspond au niveau d'interruption, la deuxième au nombre d'interruptions reçues. Certains niveaux d'interruption sont affectés de manière statique pour une architecture donnée. Ces valeurs étant valables depuis la nuit des temps, soit l'époque MS-DOS. Dans le cas du x86, on peut citer :

- niveau 0 pour le compteur système;
- niveau 1 pour le clavier;
- niveau 3 et 4 pour les ports série;
- niveau 7 pour le port parallèle.

Dans le cas d'autres périphériques tels que ceux connectés au bus PCI, les interruptions sont allouées dynamiquement et pourront être partagées entre les périphériques.

L'API des pilotes Linux fournit la fonction `request_irq()` qui permet d'associer un niveau d'interruption à une fonction de traitement (ou *handler* d'interruption). L'appel à `request_irq()` s'effectue souvent dans la fonction d'initialisation du module.

```
int request_irq(unsigned int irq, void (*gest)(int, vo
id *, struct pt_regs *), unsigned long drap_interrupt,
const char *devname, void *dev_id);
```

Les paramètres utilisés sont les suivants :

- Niveau d'interruption.
- Pointeur sur la fonction de traitement de l'interruption.
- Un drapeau indiquant le mode de traitement de l'interruption. En général, on utilise `SA_INTERRUPT` pour indiquer un traitement rapide de l'interruption (exemple : interruption compteur) ou `SA_SHIRQ` pour permettre le partage des interruptions (comme dans le cas de périphériques PCI). Dans le cas du traitement rapide, tous les niveaux d'interruption sont masqués durant l'exécution de la fonction de traitement. Dans les autres cas, seul le niveau courant est masqué.
- Une chaîne de caractères correspondant à la valeur affichée dans `/proc/interrupts`.
- Un pointeur identifiant l'interruption. En général, on répète le pointeur sur la fonction de traitement. Dans le cas d'une interruption partagée, ce paramètre est nécessaire pour identifier le source de l'interruption.

Pour libérer le niveau d'interruption, on utilise la fonction `free_irq()`. Celle-ci est souvent appelée dans la fonction de libération du module.

```
void free_irq(unsigned int irq, void *dev_id);
```

Les paramètres utilisés sont les suivants :

- niveau d'interruption ;
- un pointeur identifiant l'interruption ou bien le pointeur `NULL`.

Le code suivant permet de définir la fonction `irq1_interrupt()` au niveau d'interruption numéro 10.

```
#include <linux/interrupt.h>
int irq = 10;
/* ... */
/* Request the IRQ */
result = request_irq(irq, irq1_interrupt, SA_INTERRUPT, "irq1", irq1_interrupt);
if (result) {
        printk(KERN_ERR "irq1_init: can't get assigned irq %i\n", irq);
        return result;
}
```

La libération du niveau 10 s'effectue par le code suivant :

```
free_irq (irq, irq1_interrupt);
```

Le code de la fonction de traitement est le suivant. Il est nécessaire de retourner la valeur `IRQ_HANDLED` si l'interruption a été traitée correctement, `IRQ_NONE` dans le cas contraire.

```
static irqreturn_t irq1_interrupt (int irq, void
*dev_id, struct pt_regs *regs)
{
  printk(KERN_INFO "irq1_interrupt: got interrupt\n");
  return IRQ_HANDLED;
}
```

Le code présenté ci-dessus n'est cependant pas utilisable dans le cas où le traitement de l'interruption nécessite une faible latence. La figure 1 ci-dessous décrit schématiquement le traitement d'une interruption sous Linux en indiquant les différentes sources de latence (schéma de Michael Opdenacker).

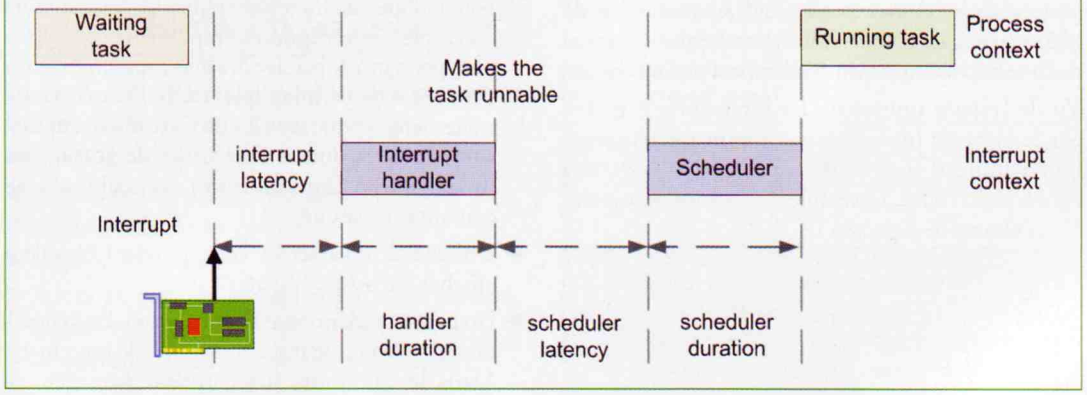
REMARQUE

Il n'est pas possible de restituer correctement le pilote d'interruption standard et le PC devra être redémarré pour fonctionner correctement. Pour assurer un redémarrage correct, il faudra soit y accéder par réseau, soit lancer le script suivant dans une autre console :

```
# sleep 300; reboot
```

Ce script nous laisse 5 minutes pour effectuer le test puis redémarre « proprement » le PC.

Figure 1 : Traitement d'une interruption



Le schéma permet de constater que les sources de latence sont nombreuses. Il est donc fréquent (mais pas obligatoire) de séparer le traitement en deux parties.

- Le traitement immédiat de l'interruption (appelé *top-half*). Cette fonction doit être la plus courte possible afin de libérer le niveau au plus tôt. Le plus souvent, on se contentera – si cela est nécessaire – d'acquitter l'interruption au niveau matériel. Ce point dépend du matériel, mais en général, cela peut être fait en lisant le contenu d'une adresse mémoire. La suite de la fonction consiste à démarrer l'exécution de la deuxième partie du traitement.
- Le traitement décalé de l'interruption (appelé *bottom-half*). Dans cette partie, on utilisera soit une file de travail (voir paragraphe précédent), soit une *tasklet* traitée par l'ordonnanceur. Dans les deux cas, le traitement est asynchrone par rapport au contexte de l'interruption (une autre interruption peut alors être prise en compte).

Il n'est pas aisé d'illustrer la démonstration par un exemple simple. Aussi, nous allons utiliser le niveau d'interruption numéro 1 correspondant au clavier du PC. Le principe est de remplacer la fonction de traitement standard par une nouvelle fonction basée sur une file de travail ou bien une tasklet.

Comme précédemment nous devons déclarer la file de travail au début du module.

```
#define MY_WORK_QUEUE_NAME "WQsched.c"
static struct workqueue_struct *my_workqueue;
```

La fonction `init_module()` doit tout d'abord supprimer l'ancienne fonction de traitement afin d'installer la nouvelle, d'où l'utilisation préliminaire de `free_irq()`.

```
int init_module()
{
     my_workqueue = create_workqueue(MY_WORK_QUEUE_NAME);
     free_irq(1, NULL);
     return request_irq (1,    /* The number of the keyboard IRQ on PCs */
                         irq_handler,      /* our handler */
                         SA_SHIRQ, "test_keyboard_irq_handler",
                         (void *)(irq_handler));
}
```

La fonction `cleanup_module()` se contente de libérer le niveau 1.

```
void cleanup_module()
{
          free_irq(1, NULL);
}
```

La fonction de traitement d'interruption place immédiatement la tâche de lecture du caractère associée à l'interruption dans la file de travail à traiter par l'ordonnanceur grâce à la fonction `queue_work()`.

```
irqreturn_t irq_handler(int irq, void *dev_id, struct
pt_regs *regs)
```

```
{
    static int initialised = 0;
    static unsigned char scancode;
    static struct work_struct task;
    unsigned char status;

    /*
     * Read keyboard status
     */
    status = inb(0x64);
    scancode = inb(0x60);

    if (initialised == 0) {
        INIT_WORK(&task, got_char, &scancode);
        initialised = 1;
    } else {
        PREPARE_WORK(&task, got_char, &scancode);
    }

    queue_work(my_workqueue, &task);

    return IRQ_HANDLED;
}
```

Dans le cas de l'utilisation d'une tasklet, le code est très similaire, bien qu'un peu plus simple.

```
static void got_char(void *dummy)
{
    static unsigned char scancode;
    unsigned char status;

    status = inb(0x64);
    scancode = inb(0x60);

    printk(KERN_INFO "Scan Code %x %s.\n",scancode &
0x7F, scancode & 0x80 ? "Released" : "Pressed");
}

DECLARE_TASKLET(intprt2_tasklet, got_char, 0);

irqreturn_t irq_handler(int irq, void *dev_id, struct
pt_regs *regs)
{
    /* Schedule the tasklet */
    tasklet_schedule(&intprt2_tasklet);

    return IRQ_HANDLED;
}
```

Le test du module est visible dans le fichier `/var/log/messages`.

```
Feb 11 22:37:39 dhcp-666-298 kernel: Trying to free free IRQ1
Feb 11 22:37:39 dhcp-666-298 kernel: Scan Code 1c Released.
Feb 11 22:37:45 dhcp-666-298 kernel: Scan Code 10 Pressed.
Feb 11 22:37:45 dhcp-666-298 kernel: Scan Code 10 Released.
Feb 11 22:37:45 dhcp-666-298 kernel: Scan Code 30 Pressed.
Feb 11 22:37:45 dhcp-666-298 kernel: Scan Code 30 Released.
Feb 11 22:37:45 dhcp-666-298 kernel: Scan Code 2e Pressed.
Feb 11 22:37:45 dhcp-666-298 kernel: Scan Code 2e Released.
Feb 11 22:37:46 dhcp-666-298 kernel: Scan Code 20 Pressed.
Feb 11 22:37:46 dhcp-666-298 kernel: Scan Code 20 Released.
```

3. Les threads du noyau

L'utilisation des processus légers (ou threads) est largement répandue de nos jours. Dans l'espace utilisateur, on utilise le plus souvent les threads POSIX. Une introduction à ce type de programmation avait fait l'objet d'un article dans *GNU/Linux Magazine* en 1999 et la référence est reprise dans la bibliographie.

Un thread dans l'espace du noyau a une structure similaire à celle d'un thread en espace utilisateur, mis à part que son contexte d'exécution est plus léger et donc moins consommateur de ressources. Dans le cas du noyau Linux, les threads sont utilisés pour des tâches récurrentes effectuées en « arrière-plan ». On peut citer la gestion de la mémoire virtuelle ou bien le traitement des journaux de certains systèmes de fichiers tels que EXT3.

Il existe assez peu de littérature sur la programmation des threads du noyau. Au moment de la rédaction de cet article, il existe deux manières de programmer des threads du noyau.

- utiliser l'API classique basée sur la fonction `kernel_thread()`.
- utiliser la nouvelle API simplifiée disponible depuis la version 2.6.14 du noyau via les fonctions `kthread_run()` et `kthread_stop()`.

Dans la suite du paragraphe, nous allons décrire un exemple de chaque API. L'exemple présenté crée deux threads identifiés par A et B, chacun affichant un message toutes les secondes.

Dans le cas de l'API classique, on doit tout d'abord déclarer les identifiants des threads. On doit également déclarer une fonction de terminaison nommée ici `on_exit`.

```
static int kthreadA_id, kthreadB_id;
static DECLARE_COMPLETION(on_exit);
```

Dans la fonction d'initialisation du module, on crée le thread en lui associant un pointeur de fonction. Dans notre cas, nous utilisons la même fonction, mais nous passons en paramètre le nom du thread (`kthreadA` ou `kthreadB`). Il faut noter qu'en cas d'erreur sur la création du deuxième thread, on arrête le premier thread en utilisant la fonction `kill_proc()`. On attend la fin du thread grâce à la fonction `wait_for_completion()`.

```
static int __init kthread1_init(void)
{
  if (!(kthreadA_id = kernel_thread (kthread_func, "kthreadA", CLONE_KERNEL)))
    return -EIO;
  if (!(kthreadB_id = kernel_thread (kthread_func, "kthreadB", CLONE_KERNEL))) {
    kill_proc (kthreadA_id, SIGTERM, 1);
    wait_for_completion (&on_exit);
    return -EIO;
  }
  return 0;
}
```

Dans la fonction d'arrêt du module, on utilise les mêmes fonctions pour les deux threads.

```
static void __exit kthread1_exit(void)
{
  kill_proc (kthreadA_id, SIGTERM, 1);
  kill_proc (kthreadB_id, SIGTERM, 1);
  wait_for_completion (&on_exit);
  wait_for_completion (&on_exit);
}
```

La fonction associée au thread est une boucle infinie. L'appel à la fonction `daemonize()` libère les ressources associées à la création du thread. La fonction `allow_signal()` permet au thread de recevoir le signal de terminaison `SIGTERM`.

Nous provoquons l'arrêt automatique des threads après 20 secondes de fonctionnement ou bien lors de la réception du signal SIGTERM au déchargement du module. Enfin, on signale la fin de l'exécution du thread grâce à la fonction complete_and_exit().

```
static int kthread_func (void *data)
{
  int nrun = 0;
  char *s = (char*)data;
  printk (KERN_INFO "%s starting\n", s);
  daemonize (s);
  allow_signal (SIGTERM);
  while( 1 ) {
    ssleep (1);
    nrun++;
    printk (KERN_INFO "%s running %d\n", s, nrun);
    if (signal_pending (current) || nrun == 20)
      break;
  }
    printk (KERN_INFO "%s exiting\n", s);
  complete_and_exit (&on_exit, 0);
}
```

Après compilation et insertion du module par insmod, on peut visualiser les threads en cours d'exécution grâce à la commande ps.

```
# insmod kthread1.ko
# ps waux | grep kthread
root         6  0.0  0.0      0     0 ?    S<   15:43   0:00 [kthread]
root      3096  0.0  0.0      0     0 ?    D    16:34   0:00 [kthreadA]
root      3097  0.0  0.0      0     0 ?    D    16:34   0:00 [kthreadB]
```

On visualise les traces dans /var/log/messages.

```
# tail /var/log/messages
Feb 17 15:57:23 dhcp-671-58 kernel: kthreadA starting
Feb 17 15:57:23 dhcp-671-58 kernel: kthreadB starting
Feb 17 15:57:24 dhcp-671-58 kernel: kthreadA running 1
Feb 17 15:57:24 dhcp-671-58 kernel: kthreadB running 1
Feb 17 15:57:25 dhcp-671-58 kernel: kthreadA running 2
Feb 17 15:57:25 dhcp-671-58 kernel: kthreadB running 2
Feb 17 15:57:26 dhcp-671-58 kernel: kthreadA running 3
Feb 17 15:57:26 dhcp-671-58 kernel: kthreadB running 3
...
Feb 17 16:53:14 dhcp-671-58 kernel: kthreadB running 19
Feb 17 16:53:15 dhcp-671-58 kernel: kthreadA running 20
Feb 17 16:53:15 dhcp-671-58 kernel: kthreadA exiting
Feb 17 16:53:15 dhcp-671-58 kernel: kthreadB running 20
Feb 17 16:53:15 dhcp-671-58 kernel: kthreadB exiting
```

Dans le cas de la nouvelle API, on déclare les threads de la manière suivante. Notez qu'il est nécessaire d'inclure le fichier d'en-tête kthread.h.

```
#include <linux/kthread.h>
struct task_struct *kthreadA_id;
struct task_struct *kthreadB_id;
```

La fonction d'initialisation du module est simplifiée, car elle se résume à l'appel de la fonction kthread_run().

```
static int __init kthread2_init(void)
{
  if (!(kthreadA_id = kthread_run (kthread_func, "A", "kthreadA")))
 if (!(kthreadB_id = kthread_run (kthread_func, "B", "kthreadB"))) {
    kthread_stop (kthreadA_id);
    return -EIO;
  }
  return 0;
}
```

Idem pour la fonction d'arrêt du module, qui est limitée à l'appel à la fonction kthread_stop().

```
static void __exit kthread2_exit(void)
{
  kthread_stop (kthreadA_id);
  kthread_stop (kthreadB_id);
}
```

La fonction associée au thread a une structure similaire à celle de l'exemple précédent, mais sa structure est également plus simple. Elle est constituée d'une boucle infinie attendant la terminaison du thread grâce à la fonction kthread_should_stop().

```
static int kthread_func (void *data)
{
  int nrun = 0;
  char *s = (char*)data;
    printk (KERN_INFO "kthread %s starting\n", s);
  do {
    ssleep (1);
    nrun++;
    printk (KERN_INFO "kthread %s running %d\n", s, nrun);
  } while (!kthread_should_stop ());
    printk (KERN_INFO "kthread %s exiting\n", s);
  return 0;
}
```

Dans cet exemple, nous n'avons pas prévu d'arrêt automatique. La fin des threads est provoquée par le déchargement du module.

```
# insmod kthread2
# tail -f /var/log/messages
Feb 17 17:02:25 dhcp-671-58 kernel: kthread A running 15
Feb 17 17:02:25 dhcp-671-58 kernel: kthread B running 15
Feb 17 17:02:26 dhcp-671-58 kernel: kthread A running 16
Feb 17 17:02:26 dhcp-671-58 kernel: kthread B running 16
Feb 17 17:02:27 dhcp-671-58 kernel: kthread A running 17
Feb 17 17:02:27 dhcp-671-58 kernel: kthread B running 17
...
# rmmod kthread2
# tail -f /var/log/messages
Feb 17 17:02:40 dhcp-671-58 kernel: kthread B running 30
Feb 17 17:02:41 dhcp-671-58 kernel: kthread A running 31
Feb 17 17:02:41 dhcp-671-58 kernel: kthread B running 31
Feb 17 17:02:42 dhcp-671-58 kernel: kthread A running 32
Feb 17 17:02:42 dhcp-671-58 kernel: kthread B running 32
Feb 17 17:02:43 dhcp-671-58 kernel: kthread B running 33
Feb 17 17:02:43 dhcp-671-58 kernel: kthread A running 33
Feb 17 17:02:43 dhcp-671-58 kernel: kthread A exiting
Feb 17 17:02:44 dhcp-671-58 kernel: kthread B running 34
Feb 17 17:02:44 dhcp-671-58 kernel: kthread B exiting
```

Dans le cas d'un système multiprocesseur, on peut également affecter un thread à un processeur donné grâce à la fonction kthread_bind().

```
void kthread_bind(struct task_struct *k, unsigned int cpu);
```

4. L'accès au matériel : la méthode mmap()

Nous avons déjà effectué des accès à des ports d'entrée/sortie dans l'article précédent concernant les pilotes en mode caractère. Nous rappelons qu'il est possible d'accéder à un port (exemple : le port parallèle du PC) en utilisant les fonction inb() et outb().

```
unsigned char inb (unsigned short port);
void outb (unsigned short port, void* addr, unsigned long count);
```

De même, un pilote peut réserver puis libérer un ensemble de ports en utilisant les fonctions `request_region()` et `release_region()`.

```
unsigned char inb (unsigned short port);
void outb (unsigned short port, void* addr, unsigned long count);
```

Nous rappelons que la liste des ports est disponible dans le fichier `/proc/ioports`.

```
# cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
01f0-01f7 : ide0
02f8-02ff : serial
0378-037a : parport0
...
```

Cette méthode est souvent utilisée pour lire ou écrire des octets au travers de registres depuis ou vers des périphériques (exemple : un UART). Dans ce cas, on utilisera les méthodes `read()` ou `write()` des pilotes, ces dernières effectuant des appels `inb()` ou `outb()`. Dans d'autres cas – comme les cartes d'acquisition vidéo – il est nécessaire de transférer rapidement de gros volumes de données et cela n'est pas possible en utilisant `read()` et `write()`.

L'API du noyau Linux fournit une méthode spéciale appelée `mmap()` permettant « mapper » un espace de mémoire et de le rendre directement accessible depuis un programme utilisateur. Vu de l'espace utilisateur, la mémoire sera considérée comme un pointeur de données. L'exemple présenté est donc constitué d'un module noyau et d'un programme utilisateur exploitant ce module.

Dans la fonction d'initialisation du module, on alloue dynamiquement une zone mémoire à l'aide de la fonction `kmalloc()`.

```
buffer = kmalloc(MMT_BUF_SIZE,GFP_KERNEL);
if (!buffer) {
                printk(KERN_INFO "failed kmalloc\n");
                unregister_chrdev (ret, "mmaptest");
                return 0;
}
```

On doit ensuite convertir l'adresse virtuelle allouée en adresse de page et réserver cette zone de mémoire. On utilise pour cela `virt_to_page()` et `SetPageReserved()`.

```
for (page = virt_to_page(buffer); page < virt_to_page(buffer
+ MMT_BUF_SIZE); page++) {
            SetPageReserved(page);
}
```

On peut ensuite copier la chaîne de test dans le tampon.

```
strcpy (buffer,"This is a mmaptest");
```

Lors du déchargement du module, on doit libérer l'espace de mémoire alloué grâce à l'appel à `ClearPageReserved()`. On peut alors libérer le tampon par `kfree()`.

```
for (page = virt_to_page(buffer); page < virt_to_page(buffer
+ MMT_BUF_SIZE); page++) {
            ClearPageReserved(page);
}
kfree(buffer);
```

La méthode `mmap()` en tant que telle est décrite par le code suivant. On doit « mapper » l'adresse du tampon sur l'adresse virtuelle du processus. Pour cela, on utilise les fonctions `virt_to_phys()`, puis `remap_pfn_range()`.

```
static int mmaptest_mmap(struct file *filp, struct vm_area_struct *vma)
{
      unsigned long page,pos;
          unsigned long start = (unsigned long)vma->vm_start;
          unsigned long size = (unsigned long)(vma->vm_end-vma->vm_start);

          printk(KERN_INFO "mmaptest_mmap called\n");

          /* if userspace tries to mmap beyond end of our buffer, fail */
      if (size>MMT_BUF_SIZE)
            return -EINVAL;

          /* start off at the start of the buffer */
      pos = (unsigned long) buffer;

          /* loop through all the physical pages in the buffer */
          /* Remember this won't work for vmalloc()d memory ! */
      while (size > 0) {
                    /* remap a single physical page to the process's vma */
            page = virt_to_phys((void *)pos);
                    if (remap_pfn_range(vma, start, page >> PAGE_SHIFT,
PAGE_SIZE, PAGE_SHARED))
                                return -EAGAIN;

            start+=PAGE_SIZE;
            pos+=PAGE_SIZE;
            size-=PAGE_SIZE;
        }

        return 0;
}
```

Du coté du programme utilisateur, on ouvre le fichier spécial associé au module, puis on utilise l'appel système `mmap()` pour « mapper » la mémoire du module sur un pointeur.

```
char *str;
fd = open(argv[1],O_RDONLY);
str = (char *) mmap (NULL, MMT_BUF_SIZE, PROT_READ,
MAP_PRIVATE, fd, 0);
if (str == MAP_FAILED) {
          perror("mmaptest user ");
          return 1;
};
/* there you go */
printf ("%s\n", str);
munmap(str, MMT_BUF_SIZE);
close(fd);
```

Lors de l'exécution, on obtient le résultat suivant. On doit créer le nœud de majeur 253 qui est alloué dynamiquement.

```
# insmod mmap_test.ko
# mknod /dev/chardev_test c 253 0
# ./mmapdriver /dev/chardev_test
This is a mmaptest
```

Cette méthode est utilisée dans l'API *Video For Linux* du noyau (V4L et V4L2). Un exemple de pilote V4L

simple a déjà été présenté dans *GNU/Linux Magazine* en mars 2005. Un exemple plus complet basé sur la nouvelle API V4L2 (ViVi pour *Virtual Video driver*) est disponible dans les nouvelles versions du noyau 2.6 dans le répertoire *drivers/media/video* des sources ou bien sur le site *Video Technology* (voir bibliographie). Un article à jour concernant l'API V4L2 sera publié ultérieurement dans *GNU/Linux Magazine*.

5. Synchronisation par sémaphores et « mutex »

Les techniques de synchronisation sont largement utilisées pour la programmation *multithread* POSIX en espace utilisateur. Dans le cas des sémaphores, on peut citer les fonctions suivantes :

```
int sem_init(sem_t *sem, int pshared, unsigned int
valeur);
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

Pour les mutex, on peut citer les fonctions suivantes :

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Dans le cas de la programmation en espace noyau, on distingue deux API dont l'une est disponible uniquement depuis la version 2.6.16. Le but est le plus souvent de protéger l'accès à une ressource matérielle pour le temps d'une configuration ou d'une lecture d'état.

Pour l'ancienne API, on doit tout d'abord déclarer l'en-tête associé :

```
# include <asm/semaphore.h>
```

La déclaration, puis l'utilisation, du sémaphore s'effectuent comme suit :

```
static DECLARE_MUTEX(my_sem);
...
down (&my_sem); /* je prends la ressource si elle est disponible */
/* J'utilise la ressource */
up (&my_sem); /* Je libère la ressource */
```

Dans le cas de la nouvelle API, on utilise le code suivant :

```
# include <asm/semaphore.h>
...
static DEFINE_MUTEX(my_sem);
...
mutex_lock (&my_sem); /* je prends la ressource si elle est disponible */
/* J'utilise la ressource */
mutex_unlock (&my_sem); /* Je libère la ressource */
```

Nous pouvons constater l'évolution du code source du fichier `drivers/media/video/videodev.c` entre la version 2.6.14 et la version 2.6.18. Dans la version 2.6.14, nous avons :

```
static DECLARE_MUTEX(videodev_lock);
...
/*
 *      Open a video device.
 */
static int video_open(struct inode *inode, struct file *file)
{
        unsigned int minor = iminor(inode);
        int err = 0;
        struct video_device *vfl;
        struct file_operations *old_fops;
        if(minor>=VIDEO_NUM_DEVICES)
                return -ENODEV;
        down(&videodev_lock);
        ...
        up(&videodev_lock);
        return err;
}
```

Dans la version 2.6.18, nous avons :

```
static DEFINE_MUTEX(videodev_lock);
...
/*
 *      Open a video device - FIXME: Obsoleted
 */
static int video_open(struct inode *inode, struct file *file)
{
        unsigned int minor = iminor(inode);
        int err = 0;
        struct video_device *vfl;
        const struct file_operations *old_fops;
        if(minor>=VIDEO_NUM_DEVICES)
                return -ENODEV;
        mutex_lock(&videodev_lock);
        ...
        mutex_unlock(&videodev_lock);
        return err;
}
```

BIBLIOGRAPHIE

- *The Linux kernel module programming guide* sur **http://www.tldp.org/LDP/lkmpg/2.6/html/index.html**
- RUBINI (Alessandro), *Linux device drivers*, 2ème édition, sur **http://www.xml.com/ldd/chapter/book**
- FICHEUX (Pierre), « API des modules Linux », *Linux Magazine* numéro 88.
- « *Linux kernel threads in device drivers* » sur **http://www.scs.ch/~frey/linux/kernelthreads.html**
- Support de cours Open Wide « Noyau Linux » et exemples par Stelian Pop.
- Article sur l'API V4L sur **http://pficheux.free.fr/articles/lmf/v4l/video4linux_img_final.pdf**
- Le pilote VIVI sur **http://v4l.videotechnology.com/vivi.html**
- LWN: remap_pfn_page() sur **http://lwn.net/Articles/104333**
- Code source des exemples présentés dans cet article sur **http://pficheux.free.fr/articles/lmf/kernel_programming/adv_drivers/exemples.tgz**

Pierre Ficheux,

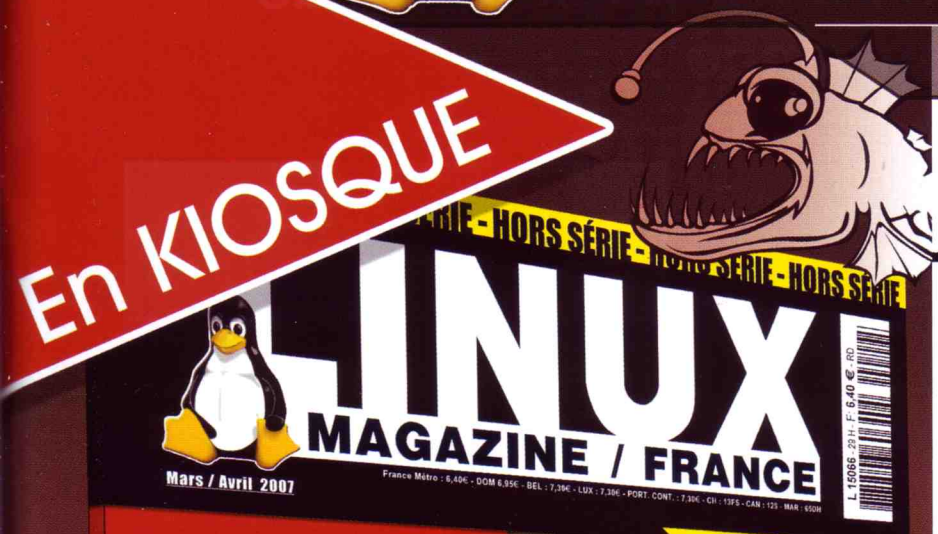
pierre.ficheux@openwide.fr

Disponible chez votre
marchand de journaux
et sur
http://www.ed-diamond.com

GNU LINUX MAGAZINE / FRANCE

HORS SERIE 29

En KIOSQUE



SOMMAIRE:

100% PINPIN