MARS 2009 N°114

GNU LINUX MAGAZINE / FRANCE

R300

Administration et développement sur systèmes UNIX

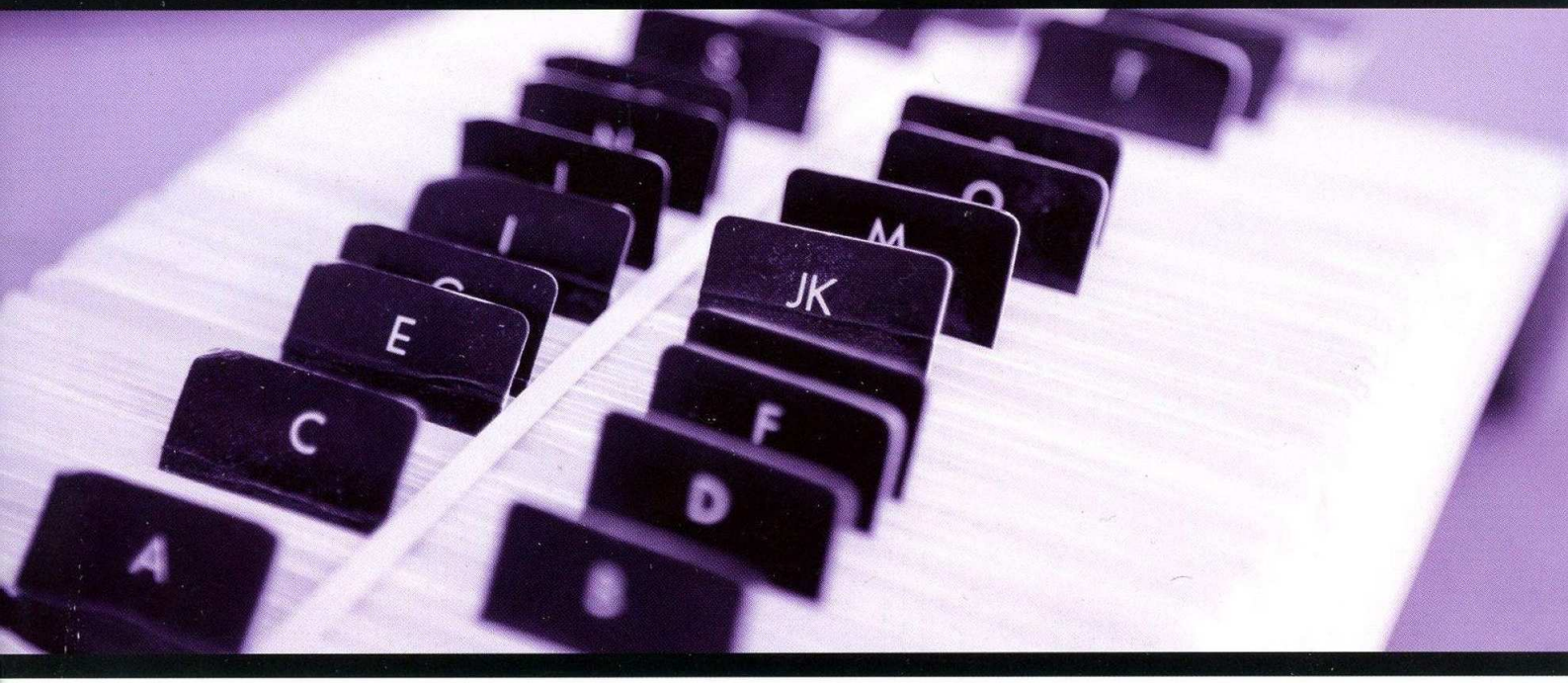
EMBARQUÉ

Installez GNU/Linux sur votre PlayStation Portable (PSP) (p. 30)

DÉCOUVRIR, INSTALLER ET CONFIGURER

PASSEZ À ZFS !

LE SYSTÈME DE FICHIERS RÉVOLUTIONNAIRE



DOMOTIQUE

DOMOGIK : commandez votre maison avec des logiciels libres

(p. 42)

CODE/SCRIPT

Découvrez toutes les ressources Python pour le Web avec Zope3, Pylons et Django

(p. 66)

NOYAU

Comprenez et utilisez l'appel système PTRACE pour analyser vos applications

(p. 04)

France Métro : 6,50 € - DOM : 7,00 €
TOM Surface : 950 XPF
POL. A : 1400 XPF
BEL/PORT.CONT : 7,50 €
CH : 13,8 CHF
CAN : 13 $CAD
MAR : 75 MAD

En panne ?

“ Nos solutions professionnelles
de redondance et de clustering,
peuvent vous aider à élaborer
un plan de reprise d'activité efficace ... ”

http://www.sivit.fr/hd

Sivit

Sommaire

Édito

De la réduction de la diversité dans l'écosystème du logiciel libre

« Voilà un titre qui présage de plein de choses », allez-vous me dire. Le genre de phrase que l'on lancera sans problème en réunion entre un « pérenniser l'efficience *open source* » et un « *business process management* à fort potentiel ». Et pourtant...

La réalité est bel et bien là. Je m'en suis rendu compte en parcourant les archives FTP de ibiblio.org et d'autres serveurs. Ces noms ne vous disent peut-être pas grand-chose : twm, fvwm, OLwm, AfterStep, wm2... Ce sont des gestionnaires de fenêtres. Subitement, je me suis rendu compte d'une réalité troublante en cherchant à répondre à une question simple : qu'est-ce que GNU/Linux aujourd'hui pour les nombreux utilisateurs qui sont venus grossir nos rangs ?

La réponse tient malheureusement souvent en une poignée de noms de projets : Ubuntu, GNOME, OpenOffice.org, The Gimp et l'incontournable duo FireFox/Thunderbird. On peut ajouter pour les plus aventureux, Debian, KDE, KOffice, Fedora, Midori... Et pour les plus « *Old School* », Enligthenment, WindowMaker, Awesome, Irssi, Vim, Mutt, Emacs, Pine...

Mais, le fait est là. De nombreux projets n'avancent plus ou peinent à le faire faute de moyens et de main d'œuvre. L'écrasante popularité des applications phares entraîne un choix quasi automatique pour les *toolkits* associés. Qu'allez-vous choisir si vous devez développer une application graphique ? GNOME/GTK ? KDE/QT ? Quoi d'autre ? Une myriade d'applications viennent donc grossir des environnements graphiques déjà dominants et ainsi renforcer encore davantage leur position.

Vous en doutez ? Où est passé le support pour Xubuntu ? Depuis quand n'avez-vous pas vu fonctionner un autre navigateur que FireFox (sur un UNIX) ? Quelle application avez-vous lancée ces 12 derniers mois pour retoucher une photo ?

La popularité d'un système GNU/Linux construite autour de quelques applications n'est structurellement et fonctionnellement pas meilleure qu'un système propriétaire. Je ne parle pas, bien entendu, des avantages incontestables du logiciel libre, mais du manque de libertés induit par l'effet de masse. Pourquoi des utilisateurs continuent-ils de pirater MS/Office, alors qu'il existe une solution gratuite et open source pour leur système propriétaire ? Par habitude, bien sûr !

Cette centralisation autour de certaines distributions et environnements m'inquiète. Elle tue la diversité et finira peut-être par priver les utilisateurs de logiciels libres de leur plus importante liberté : le choix.

Essayer de faire comprendre aux nouveaux utilisateurs qu'il existe des alternatives dans l'alternative n'est pas difficile. Ceci leur montrera qu'il existe quelque chose après leur première expérience. D'autres manières d'utiliser son système. D'autres manières de concevoir son utilisation. Qu'il ne s'agit pas juste d'un pas, mais d'un chemin à arpenter...

Un grand nombre de projets méritent qu'on s'y attarde. Ceci est tout aussi vrai pour les applications graphiques que pour les solutions serveur. Ce n'est pas parce qu'Apache représente presque 50% des serveurs Web que je regrette mon choix personnel de Lighttpd (et ses 0.21%), car il est adapté à mes besoins. Je vous conseille d'en faire autant.

Mais nous aurons le loisir de discuter de tout cela si vous êtes de passage à Paris les 31 mars et 1/2 avril prochains pour *Solution(s) (GNU/)Linux opensource*, sur le stand G30.

Nous nous retrouverons, de toutes façons, le 28 mars pour le numéro 115.

Denis Bodor

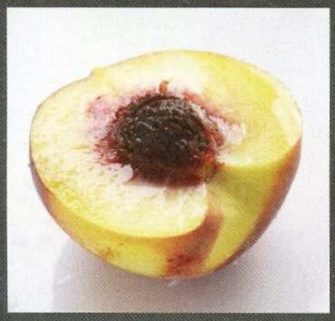
Gnu /Linux Magazine France

LES ÉDITIONS DIAMOND

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 rédaction : Véronique Wilhelm
Relecture : Dominique Grosse
Conception graphique : Fabrice Krachenfels
Responsable publicité : Tél. : 03 88 58 02 08
Service abonnement : Tél. : 03 88 58 02 08
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 : À parution /N° ISSN : 1291-78 34
Commission paritaire : K78 976
Périodicité : Mensuel
Prix de vente : 6,50 €

PTRACE : Introspection à



Que ce soit par nécessité (votre programme se comporte étrangement) ou pour parfaire vos connaissances (apprendre, c'est un petit peu le point commun de tous les lecteurs de ce magazine), chercher à comprendre les interactions qui existent entre un programme et le noyau Linux est une étape importante dans la vie du programmeur Linux. Dépositaire de ce savoir, vous serez alors capable de développer des applications permettant de déboguer des programmes, mais aussi de tracer plus simplement les appels système émis par vos applications.

Auteur

- Lionel Tricon

L'appel système **PTRACE** permet à un processus appelant d'observer et surtout de contrôler l'exécution d'un autre processus afin, par exemple, de consulter son image mémoire ou d'éditer au passage ses registres (précision utile : Cet article va se focaliser sur l'architecture **x86**).

PTRACE (pour **process trace**) est utilisé principalement par les débogueurs comme **gdb** pour positionner des points d'arrêts et suivre pas à pas l'exécution d'un programme. Mais, il est aussi utilisé par des programmes comme **strace** **[1]** ou **ltrace** **[2]** permettant, en ce qui concerne **strace**, de tracer tous les appels système utilisés par un programme et, en ce qui concerne **ltrace**, de tracer les appels aux bibliothèques partagées comme la **glibc** (de façon accessoire, **PTRACE** peut être utilisé pour développer des **rootkits** ou des injections de code, mais c'est une autre histoire).

Pour ce qui nous concerne, nous allons uniquement nous atteler à intercepter les appels système émis vers le noyau, afin de comprendre ce qui se passe dans les couches basses du système (à la lisière du noyau plus précisément).

Un prochain article traitera plus particulièrement d'une introduction aux utilitaires **strace** et **ltrace** (sachant, et c'est là tout l'intérêt, que les deux reposent sur **PTRACE**).

1 La théorie

En préalable à l'étude de l'appel système **PTRACE**, il faut d'abord nous familiariser avec la notion d'« appel système ». Une piqûre de rappel ne fait jamais de mal.

Les appels systèmes (en anglais, *system call*, abrégé en **syscall**) sont, pour ceux qui l'ignorent, des points d'entrée logiciels dans le noyau comme les **open(2)**, **close(2)** ou autre **write(2)** qui devraient vous être bien connus (des primitives d'accès en somme).

Par exemple, la routine **sys_write()** qui implémente l'appel système **write(2)** est définie de la façon suivante dans le noyau :

```
int sys_write(unsigned int fd, const char *buf,
unsigned int count);
```

Lorsqu'un processus veut invoquer un appel système (plus de 300 appels système à cette heure), il place les arguments de l'appel système dans des registres et fait appel à l'interruption 0x80 qui est la clef qui ouvre les portes du noyau (en réalité, fait basculer le processeur dans le noyau en mode superviseur). Sous une architecture i386, on utilise le registre %eax pour indiquer le numéro de l'appel système.

Dans le détail, une architecture i386 dispose d'un nombre limité de registres généralistes de taille 32 bits avec dans l'ordre EAX, EBX, ECX, EDX, ESI et EDI qui peuvent être utilisés pour une multitude de choses et en particulier l'accès aux appels système sous Linux par l'interruption **0x80** (par convention).

Pour illustrer cela, voici un exemple concret d'invocation d'appels système en se limitant aux quatre registres les plus usités.

Le formalisme retenu est de présenter la liste des arguments de l'appel système en première ligne, puis la valeur de retour (en italique) représentée sur la seconde ligne et qui prend la place du registre EAX, ce même registre EAX identifiant à l'aller l'appel système dans le noyau (par exemple, numéro 5 pour **open**).

Syscall	EAX	EBX	ECX	EDX
open	5	chemin à ouvrir	-	-
	descripteur de fichier	-	-	-
read	3	descripteur de fichier	Pointeur sur un buffer	Nombre d'octets à lire
	nombre d'octets lus	-	-	-
write	4	descripteur de fichier	Pointeur sur un buffer	Nombre d'octets à écrire
	nombre d'octets écrits	-	-	-
close	6	descripteur de fichier	-	-
	Code retour	-	-	-

La plupart des appels se contentent effectivement de cinq registres (six moins un) pour sauver leurs paramètres (les paramètres qui ne peuvent se satisfaire de registres 32 bits passent leurs valeurs par adresse comme une chaîne de caractères ou une structure).

Mais, si l'on prend en exemple l'appel système **mmap(2)**, celui-ci en a besoin de six. Avec l'identifiant de l'appel système, cela fait sept : nous avons donc un problème. Pas de panique : Dans ce cas, un registre est réservé pour pointer sur une zone mémoire de l'espace utilisateur qui contient les paramètres restants. À charge au noyau de récupérer les paramètres restant lors de ses traitements.

Et c'est ce mécanisme standard qui peut être intercepté par le noyau. Si l'on place un processus sous surveillance grâce à l'appel système PTRACE, le noyau va stopper net le déroulement du programme et donner la main au processus observeur lors de sa rencontre avec un appel système.

Le prototype de PTRACE est détaillé ci-dessous (disponible sous **<sys/ptrace.h>**) :

```
long int ptrace(enum __ptrace_request request,
                pid_t pid,
                void *addr,
                void* data);
```

L'argument **request** indique la commande à accomplir, **pid** l'identifiant du processus sous surveillance, **addr** correspond à l'offset en espace utilisateur où les données **data** seront écrites ou récupérées.

La méthodologie généralement observée est la suivante :

Le programme observeur fait appel à la primitive **fork(2)** pour dupliquer son processus et exécuter dans le fils la commande **PTRACE_TRACEME** suivie d'un **excl(3)** sur l'exécutable qu'il souhaite superviser (de façon alternative, il est possible de s'attacher à un processus en cours d'exécution par l'intermédiaire de la commande **PTRACE_ATTACH**).

Lors de la réception d'un signal ou du traitement d'un appel système qui affecte le fils (en fait, c'est aussi un signal qui est envoyé : **SIGTRAP**), le père est notifié du changement d'état grâce à **wait(2).**

Dans le père, il suffit de bloquer sur le **wait(2)** et d'attendre un événement. Dès que cela arrive, plusieurs options s'offrent à vous :

- Si c'est un signal, vous pouvez consulter la valeur du signal grâce à la commande **PTRACE_GETSIGINFO**.
- Si c'est un appel système, vous devez tout d'abord récupérer le numéro de l'appel système qui est présent dans le registre %eax grâce à la commande **PTRACE_PEEKUSER**.

Nous verrons plus loin qu'il n'est pas forcément évident de distinguer les deux.

Par la suite, il vous sera possible (dans le cas d'un appel système) d'utiliser une des primitives de la famille **PTRACE_PEEK** afin de récupérer les informations de l'appel système courant vers un emplacement mémoire (registres, informations sur le processus).

Modifier les attributs est possible en utilisant une des primitives de la famille **PTRACE_POKE** (ce qui est d'ailleurs une des fonctionnalités majeures de cette solution technique, qui offre un champ de possibilités appréciable).

Lorsque vous avez traité l'évènement, et que vous souhaitez reprendre le fil normal du processus, il faut faire appel à la commande **PTRACE_CONT** (redémarrer le processus fils arrêté).

Si vous souhaitez redémarrer le fils, mais être arrêté au prochain appel système, il faut utiliser la commande **PTRACE_SYSCALL**. De façon similaire, la commande **PTRACE_SINGLESTEP** permet de reprendre le cours du processus fils, mais d'être notifié cette fois du traitement de la prochaine instruction (très pratique pour les débogueurs).

D'autres options sont disponibles et je vous invite à consulter la page *man* si vous souhaitez en apprendre un peu plus.

2 Passons à la pratique

Comme cela est sans doute un peu obscur et théorique, passons à quelques exemples concrets permettant de mieux appréhender tous ces concepts.

Nous allons écrire un exemple qui va mettre en œuvre la primitive **ptrace(2)** et les commandes **PTRACE_TRACEME** et **PTRACE_CONT**. Notre programme va appeler la commande **/bin/ls** sur la racine et afficher la valeur du premier appel système :

```
#include <sys/ptrace.h>
#include <sys/syscall.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
#include <sys/reg.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t child;
    int status;

    if ((child = fork()) == 0)
    {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "/bin/ls", "/", NULL);
    }

    for (;;)
    {
        wait(&status);
        if(WIFEXITED(status)) break;
        long orig_eax = ptrace(PTRACE_PEEKUSER, child, 4*ORIG_EAX, NULL);
        printf("Appel système numéro : %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }

    return 0;
}
```

On retrouve le principe de *forker* la commande à exécuter dans un fils (primitives **fork** et **excl**) et à se mettre en attente des évènements dans le père.

On notera l'utilisation **WIFEXITED(status)** qui renvoie vrai si le statut provient d'un processus fils qui s'est terminé normalement.

On note aussi la référence au registre %eax (en fait sa sauvegarde par le noyau) par l'utilisation de la constante **ORIG_EAX** qui est déclarée sous **/usr/include/sys/reg.h** (convention de PTRACE). On note aussi que la valeur est multipliée par 4, ce qui est logique, car on adresse un tableau d'entier.

On compile avec **gcc** et on exécute pour obtenir la sortie suivante :

```
$ gcc -o strace strace.c
$ ./strace
Appel système : 11
bin  boot  cdrom  dev  etc  home  initrd  initrd.img  lib  lost+found
media  mnt  mount  opt  proc  root  sbin  srv  sys  tmp  usr  var  vmlinuz
```

Hum, on constate que le premier appel système (et le seul puisque l'on a choisi de poursuivre l'exécution) est 11. Pour connaître la correspondance avec l'appel système, il faut consulter le fichier **/usr/include/asm-i386/unistd.h** qui nous renseigne que le numéro 11 est attribué à **execve**.

Améliorons maintenant notre mise en observation en utilisant la commande **PTRACE_SYSCALL** :

```
for (;;)
{
    […]
    ptrace(PTRACE_SYSCALL, child, NULL, NULL);
}
```

On constate maintenant que tous les appels système sont tracés :

```
Appel système : 11
Appel système : 45
Appel système : 45
[…]
Appel système : 4
bin  boot  cdrom  dev  etc  home  initrd  initrd.img  lib  lost+found
media  mnt  mount  opt  proc  root  sbin  srv  sys  tmp  usr  var  vmlinuz
Appel système : 4
[…]
Appel système : 91
Appel système : 91
Appel système : 252
```

Vous remarquerez que les appels sont doublés. Effectivement, on est notifié lors de l'entrée dans un appel système, mais aussi lors de la sortie.

Le dernier appel système de numéro 252 est en fait l'appel système **exit_group** qui permet de terminer tous les *threads* du processus.

Prochaine étape : Afficher les paramètres d'un appel système et son code retour. On va utiliser pour cela le fichier **/usr/include/sys/syscall.h** qui déclare des constantes pour tous les appels système (**SYS_write** dans notre cas).

```
int appel_syscall = 1;
for (;;)
{
    long param[3];

    wait(&status);
    if(WIFEXITED(status)) break;
    int orig_eax = ptrace(PTRACE_PEEKUSER, child, 4*ORIG_EAX, NULL);
    if (orig_eax == SYS_write)
    {
        if (appel_syscall)
        {
            appel_syscall = 0;
            param[0] = ptrace(PTRACE_PEEKUSER, child, 4*EBX, NULL);
            param[1] = ptrace(PTRACE_PEEKUSER, child, 4*ECX, NULL);
            param[2] = ptrace(PTRACE_PEEKUSER, child, 4*EDX, NULL);
            printf("Write(%ld, %ld, %ld)\n", param[0], param[1], param[2]);
        }
        else
        {
            int eax = ptrace(PTRACE_PEEKUSER, child, 4*EAX, NULL);
            printf("Code retour = %ld\n", eax);
            appel_syscall = 1;
        }
    }
    ptrace(PTRACE_SYSCALL, child, NULL, NULL);
}
```

On affiche les paramètres de l'appel système lors de la première occurrence et le code retour lors de la seconde occurrence (cette fois dans le registre EAX, car, avec **ORIG_EAX**, on n'accède qu'à une sauvegarde du numéro de l'appel système par le noyau comme précisé précédemment).

Par exemple, cela nous donne :

```
Write(1, -1209126912, 151)
bin  boot  cdrom  dev  etc  home  initrd  initrd.img  lib  lost+found
media  mnt  mount  opt  proc  root  sbin  srv  sys  tmp  usr  var  vmlinuz
Code retour = 151
```

Le premier paramètre correspond bien au flux **stdout**, le second à l'adresse du *buffer* et le dernier au nombre de caractères écrits. On retrouve ce nombre dans la valeur de retour de l'appel système.

On aurait pu aussi choisir d'utiliser **PTRACE_GETREGS** pour copier en local les registres lors d'un appel système afin de réduire le volume de code et limiter les accès. Il faut alors déclarer le fichier d'inclusion suivant **/usr/include/sys/user.h** pour pouvoir déclarer une structure **user_regs_struct**.

```
if (appel_syscall)
{
    appel_syscall = 0;
    struct user_regs_struct registres;
```

```
    ptrace(PTRACE_GETREGS, child, NULL, &registres);
    printf("Write(%ld, %ld, %ld)\n", registres.ebx, registres.ecx, registres.edx);
}
```

Tout cela est très intéressant, mais ne s'applique qu'à de la consultation de données. Comment faire si l'on souhaite non pas consulter, mais modifier les valeurs d'un appel système avant qu'il ne soit exécuté ?

Il existe en fait deux types d'appel système : **PTRACE_POKE** si l'on veut modifier les données dans la pile et **PTRACE_SYSEMU** si l'on veut émuler l'appel système.

Pour ce dernier (disponible depuis le noyau 2.6.14), la commande est quasi exclusivement utilisée par **User-Mode Linux** (ou toute autre solution similaire de virtualisation basée sur l'escamotage de TOUS les appels système). L'appel à cette commande fait que le prochain appel système ne sera pas exécuté et que le programme a alors un contrôle total sur les valeurs à modifier et/ou à retourner (le programme se substitue au noyau).

La première commande nous est plus familière, car il s'agit uniquement de modifier les valeurs des paramètres passés au noyau ou retournés à l'appelant. C'est, par exemple, la technologie employée par **fakerootng** **[3]** qui est similaire à **chroot(2)**, mais ne nécessitant pas de privilèges *root*, car basé sur le remplacement des chemins dans certains appels système par injection de code (fonctionnellement similaire à **fakechroot** **[4]**, alors que ce dernier est basé sur le mécanisme **LD_PRELOAD**).

Prenons un cas d'utilisation assez simple où nous allons modifier la valeur du flux de sortie de la primitive **write(2)**. Notre programme va changer à la volée le flux de sortie de **stdout** vers **stderr**. Rien de bien fonctionnel, mais qui a l'avantage d'être simple à implémenter (il est plus compliqué de modifier une chaîne de caractères par exemple).

```
if (orig_eax == SYS_write)
{
    if (appel_syscall)
    {
        appel_syscall = 0;
        int value = ptrace(PTRACE_PEEKUSER, child, 4*EBX, NULL);
        if (value == 1) ptrace(PTRACE_POKEUSER, child, 4*EBX, 2);
    }
    else
    {
        appel_syscall = 1;
    }
}
```

Et voilà, comme par magie, nous venons de modifier un paramètre de notre appel système à chaud sans toucher au code de départ :

```
$ ./strace 1>/dev/null
bin  boot  cdrom  dev  etc  home  initrd  initrd.img  lib  lost+found
media  mnt  mount  opt  proc  root  sbin  srv  sys  tmp  usr  var  vmlinuz
$ ./strace 2>/dev/null
[vide]
```

3 PTRACE et les signaux

Nous avons vu que le lien entre l'observeur et l'observé passe par l'envoi de signaux. Le programme sous observation prévient le père de l'arrivée d'un événement par l'intermédiaire du signal SIGTRAP (sur les plateformes compatibles **POSIX**, SIGTRAP est le signal envoyé par les programmes lorsqu'une condition a été remplie pour les débogueurs et autres sous-systèmes relatifs à PTRACE).

La question que l'on est en droit de se poser concerne les signaux reçus par notre programme et la façon dont nous allons les différencier des appels système ; la primitive **wait(2)** faisant peu de cas de ce genre de distinction.

Heureusement, l'option **PTRACE_GETSIGINFO** (appelée juste après le **wait**) nous permet d'en apprendre un peu plus sur l'objet de la notification :

```
siginfo_t sig;
ptrace(PTRACE_GETSIGINFO, child, NULL, &sig);
```

Deux attributs nous intéressent plus particulièrement dans la structure **siginfo_t** :

- **si_signo** : Numéro du signal ;
- **si_code** : Provenance du signal (noyau ou espace utilisateur).

Le principe est donc de tester le type du signal afin de savoir si c'est un appel système ou la réception d'un signal qui a réveillé notre processus :

```
if (sig.si_code!=sig.si_signo && sig.si_signo!=SIGTRAP)
{
    ptrace(PTRACE_SYSCALL, child, NULL, sig.si_signo);
    continue;
}
```

Intervient ici un détail qui a toute son importance : le quatrième paramètre de l'option **PTRACE_SYSCALL** est ici valorisé, contrairement aux exemples précédents (**NULL**).

En fait, si ce champ est laissé vide, le signal ne sera pas traité par notre processus observé ; plus précisément, si ce champ contient autre chose que SIGSTOP, il est interprété comme un numéro de signal à délivrer au fils (sinon, aucun signal n'est délivré). On peut de cette façon contrôler si un signal envoyé au fils doit lui être délivré ou non.

Notez aussi que cela s'applique de façon similaire à **PTRACE_CONT**.

4 Se « protéger » de PTRACE

Il n'est pas toujours souhaitable qu'un programme puisse être sous contrôle de PTRACE. Pour éviter cela, plusieurs mécanismes préventifs peuvent être mis en place. Bien évidemment, il existe d'autres façons d'accéder à l'espace mémoire d'un processus, mais rien ne nous empêche d'essayer de compliquer la tâche à d'éventuels pirates susceptibles de retourner PTRACE contre vous.

PTRACE ne pouvant être associé qu'une fois avec un processus, le plus évident consiste à se superviser soi-même avec **PTRACE_TRACEME**.

Une autre option un peu plus lourde à mettre en œuvre consiste à bloquer PTRACE par ses signaux, plus précisément par le signal SIGTRAP. En effet, le mécanisme de supervision

transite par ce signal ; le rendre inopérant permet d'éviter toute forme d'intrusion avec PTRACE.

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>

int debug = 1;
int i;

void sigtrap(int sig)
{
    printf("Sigtrap received :)\n");
    debug = 0;
}

int main(int argc, char *argv[])
{
    signal(SIGTRAP, sigtrap);
    printf("Hello world\n");
    if (debug)
    {
        fprintf(stderr, "Vous n'êtes pas autorisé à faire cela ....\n");
        return -1;
    }

    return 0;
}
```

Exemple d'exécution :

```
$ gdb [programme]
GNU gdb 6.6-debian
[…]
(gdb) run
Starting program: [programme]
Hello world
Vous n'êtes pas autorisé à faire cela ....
Program exited with code 0377.
```

5 Pour conclure

PTRACE est le point de passage obligé pour qui veut développer une application qui supervise un autre programme.

Nous avons vu que son interface est loin d'être simple à mettre en œuvre et qu'il faut avoir des connaissances assez étendues pour aller plus loin qu'une simple introspection de surface et modifier le comportement d'une application.

Autre défaut de PTRACE : Ce n'est pas un appel système normalisé POSIX et son interface, peu ou mal documentée, reste donc sujet à interprétation. Son comportement varie aussi d'un système d'exploitation à l'autre et même parfois au sein des différentes versions de Linux.

Son implémentation est aussi hautement dépendante de l'architecture matérielle (ce qui rend le code dans le noyau difficile à maintenir) et du système d'exploitation. Il faut donc jongler en permanence avec les cas particuliers si l'on veut supporter plusieurs architectures lorsque l'on développe une application un tant soit peu multi-plateforme utilisant PTRACE.

Et la liste de ses défauts ne s'arrête pas là. Outre que son API n'est pas réputée pour sa simplicité, le code dans le noyau semble plutôt difficile à maintenir. Côté performance, c'est largement perfectible, car la solution actuelle pénalise les performances (induit beaucoup trop de changements de contexte dans le noyau). En résumé : tout cela aurait besoin d'une bonne remise à plat.

C'est ce qui est en train de se passer avec les patchs **Utrace** soumis par Roland McGrath en 2007 qui se propose tout simplement de remplacer littéralement PTRACE par une toute nouvelle infrastructure dans le noyau, appelée « Utrace », tout en offrant une couche de compatibilité de l'interface dans l'espace utilisateur (PTRACE devient un client de Utrace).

Utrace se veut beaucoup plus transverse que PTRACE en se concentrant sur les services offerts à l'intérieur du noyau qui permettront l'arrivée de nouvelles générations d'utilitaires de débogage ou d'introspection étendus non seulement aux programmes, mais aussi aux *drivers*, systèmes de fichiers ou autres interfaces réseau.

Utrace fonctionne sur la base de **callback** (fonctions de rappel) ; on s'abonne aux appels système ou aux signaux que l'on souhaite superviser (base de la programmation événementielle) et on accède directement aux données (consultation, modification) dans des fonctions sans avoir besoin de faire des allers-retours improductifs dans la mémoire (un fonctionnement similaire à **FUSE** par exemple). Beaucoup plus simple, intuitif et performant que l'implémentation actuelle.

Pour arriver à cela, l'infrastructure PTRACE dans les noyaux en développement est en train de subir de gros changements afin de préparer l'arrivée de Utrace (patchs **Tracehook**) **[5]** et permettre le développement à terme d'outils comme **DTRACE** **[6]** (*SUN Microsystem*) disponibles sur d'autres plateformes et qui ont dans l'immédiat une longueur d'avance.

Pour aller plus loin, je vous invite à consulter le lien **[7]** qui détaille les avantages et inconvénients de ces solutions et **[8][9]** qui décrivent des techniques d'injection de code dans des programmes en cours d'utilisation ce qui peut, à l'usage, se révéler soit très pratique (outils de sécurité ou d'introspection), soit très dangereux (développement de *rootkits*).

Liens

- **[1]** Strace : http://sourceforge.net/projects/strace/
- **[2]** Ltrace : http://sourceforge.net/projects/ltrace/
- **[3]** Fakerootng : http://sourceforge.net/projects/fakerootng
- **[4]** Fakechroot : http://fakechroot.alioth.debian.org/
- **[5]** *Utrace arch porting How-To* : http://sourceware.org/systemtap/wiki/utrace/arch/HowTo
- **[6]** DTRACE : http://opensolaris.org/os/community/dtrace/
- **[7]** « *Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps* » : http://ols.108.redhat.com/2007/Reprints/keniston-Reprint.pdf
- **[8]** « *Playing with ptrace, Part II* » : http://www.linuxjournal.com/article/6210
- **[9]** « *Playing with ptrace() for fun and profit* » : http://chdir.org/~nico/static/article-nbareil-ptrace-sstic06.pdf

Auteur : Lionel Tricon

Ingénieur caméléon spécialisé désormais dans le domaine des systèmes d'information (après quelques années passées dans le domaine des clusters HA et HPC) et surtout Linuxien et KDEiste convaincu. Réside actuellement sur Aix-en-Provence.

Le système de fichiers ext4

Le système de fichiers ext4, successeur d'ext3, est apparu en version stable dans le noyau 2.6.28 publié en décembre 2008. Il existait depuis 2 ans en version -dev, c'est-à-dire que son usage était déconseillé en production. Sa sortie en version stable indique qu'il est désormais mature pour servir au quotidien.

Auteur

- Kevin Denis

1 Introduction

Dans Unix, tout est fichier. Les fichiers sont rangés de différentes manières sur un disque selon une méthode appelée « système de fichiers » (en anglais *filesystem* ou *fs*). Il existe de nombreux systèmes de fichiers utilisables sous Linux : par exemple ext2/3, reiserfs, xfs, jfs, vfat, etc. chacun avec différentes qualités et défauts. Le système de fichiers traditionnel de Linux est ext3, successeur d'ext2.

Ext2 a été écrit de manière à pouvoir être extensible. Ext3 ajoute un journal à ext2 apportant ainsi aux systèmes de fichiers une meilleure consistance en cas de crash. Ext4 étend encore plus les possibilités des systèmes de fichiers type ext « n », au prix de modifications en profondeur sur la façon dont les données sont écrites sur le disque, offrant de meilleures performances et une meilleure vérification d'intégrité des données en cas de crash système. Les systèmes de fichiers ext « n » disposent d'un *wiki* documentaire, aujourd'hui entièrement orienté ext4 qui peut être consulté à l'adresse: http://ext4.wiki.kernel.org.

Cet article propose de passer en revue les nombreuses améliorations d'ext4 avant de proposer à titre pratique différents cas d'utilisation d'ext4.

2 Les nouveautés d'ext4

La compatibilité

Ext4 est compatible avec ext3. La compatibilité s'exprime de deux manières différentes :

Il est possible de monter une partition ext3 en ext4. Dans ce cas, aucune des améliorations d'ext4 n'est utilisée, mais il sera possible, plus tard, de remonter cette partition avec ext3.

Il est possible de migrer une partition ext3 en ext4 sans perte de données. Dès lors, il est obligatoire d'utiliser ext4 pour monter cette partition. Il devient impossible de réutiliser cette partition avec ext3 par la suite.

Certaines limites d'ext3 sont repoussées

Alors qu'ext3 limitait le nombre de sous-répertoires situés dans un répertoire à 32000, en ext4, le nombre de sous-répertoires situés dans un répertoire est illimité.

Une *inode* ext3 utilisait 128 octets. Les inodes ext4 utilisent 256 octets. Les dates de modification de fichier utilisent cet espace afin d'exprimer le « *Modified Timestamp* » en nanosecondes). De plus, les attributs étendus sont stockés directement dans l'inode, ce qui accélère les temps de traitement des programmes utilisant ces informations.

Enfin, lors de la création d'un répertoire, plusieurs inodes sont réservés contigus au répertoire afin d'accélérer la création de fichiers dans ce répertoire.

Ext3 peut utiliser une taille max de 16 To (1 teraoctet=1024 Go), et une taille maximale par fichier de 2 To. Ext4 propose une taille maximale de système de fichiers égale à 1 Eo (1 Exaoctet=1048576 To (!!)) et une taille maximale par fichier égale à 16 To. Toutefois le code actuel de création de système de fichiers ext4 supérieur à 16 To n'a pas été encore stabilisé.

Les extents

Les *extents* font partie des améliorations majeures d'ext4. En ext3 et dans les systèmes de fichiers habituels, tout fichier est découpé en blocs, chacun de ces blocs étant référencé. Pour lire un fichier volumineux, il est donc nécessaire de lire la liste des blocs, ce qui

peut être long, puis d'accéder aux données ce qui est vite pénalisant en ressource.

Le mécanisme d'extents indique que le fichier démarre au bloc « x » et continue pendant les « n » blocs suivants. La lecture est bien plus rapide. Toutefois, il faut que les blocs du fichier soient contigus, ce qui justifie les améliorations suivantes.

La défragmentation on-line

Oui, ext4 propose un mécanisme de défragmentation. Aucun système de fichiers n'échappe au phénomène de la fragmentation, certains (comme le VFAT) sont beaucoup plus sensibles que d'autres (ext2/3), car les mécanismes de gestion de l'emplacement des fichiers sont plus ou moins performants. Toutefois, ext4 étant appelé à travailler sur des systèmes possédant plusieurs gros fichiers, ce mécanisme de défragmentation a été implémenté afin qu'un fichier puisse être toujours lu de manière séquentielle.

Un outil **e4defrag** est en cours de développement. Actuellement, **e4defrag** n'est pas supporté par le noyau 2.6.28, mais le sera dans un proche futur.

Meilleure allocation de l'espace disque

Lorsqu'ext3 a besoin d'écrire un fichier, il appelle le *block allocator* qui lui indique où sont situés des blocs vides sur le disque. Cette méthode n'est pas optimale, car ext3 doit appeler les blocs un par un. Par exemple, pour un fichier de 10 Mo avec des blocs de 4 ko, ext3 doit appeler le block allocator 2560 fois !

Ext4 permet d'appeler le block allocator une unique fois pour toute taille de fichier, ce qui est plus efficace.

Ext4 utilise aussi l'allocation décalée (*delayed allocation*) qui a deux avantages. Lorsqu'une application demande à écrire des données sur le disque, ext4 attend avant d'allouer des blocs sur le disque. Si l'application continue d'écrire dans ce fichier et, de cette façon, l'étend, ext4 peut trouver un emplacement plus adéquat ayant de la place pour que le fichier reste toujours continu. Et si l'application supprime ce fichier, ext4 n'a tout simplement pas besoin de supprimer les blocs associés, ceux-ci n'ayant pas été écrits.

En ext3, les applications ayant besoin de préallouer une taille fixe (par exemple, les applications de P2P qui remplissent petit à petit un fichier) ont besoin d'écrire un fichier vide rempli de zéros sur le disque, ce qui est contreproductif.

Ext4 propose une méthode de préallocation unique permettant de disposer immédiatement le nombre de blocs contigus nécessaires. Une fois de plus, la fragmentation est évitée.

Un meilleur mécanisme de fsck et de journalisation

Les systèmes de fichiers modernes sont journalisés, ce qui permet de réduire les durées de *file system check* (**fsck**). Néanmoins, avec les tailles de disque qui augmentent, la durée de vérification recommence à devenir particulièrement longue. Ext4 utilise un mécanisme de *checksum* interne afin d'accélérer grandement les **fsck**. Un bloc non modifié ne sera donc pas vérifié lors du **fsck**.

Le journal d'un système de fichiers (comme celui d'ext3 ou ext4) est un emplacement du disque qui est utilisé le plus fréquemment. Ainsi, il est le plus directement sujet à des problèmes de hardware. Ext4 utilise un mécanisme de checksum pour vérifier son intégrité. De plus, il utilise de meilleurs algorithmes pour accélérer les traitements du journal.

Le mode « Barrier »

Les disques modernes utilisent tous différentes méthodes de mise en cache de données pour accélérer leurs traitements. Ces méthodes peuvent aller jusqu'au réordonnancement des écritures sur le disque. Dans de rares cas où un journal est utilisé, ces réordonnancements peuvent conduire à des corruptions de fichiers. Donc, une barrière logicielle est mise en place permettant d'assurer l'ordre des écritures afin que les données soient consistantes sur le disque. Ce mode a bien évidemment un impact en performances. L'équipe de développement d'ext4 conseille de ne pas utiliser les barrières pour des tests de *benchmarking*. (option **-o barrier=0** lors du montage).

3 Utiliser ext4

Deux points sont à vérifier pour pouvoir utiliser ext4 : le noyau, et les outils qui manipulent le système de fichiers. Le noyau doit être supérieur à la version 2.6.28 pour utiliser ext4.

```
kevin@zipslack:~$ cat /proc/filesystems | grep ext
  ext3
  ext2
  ext4
kevin@zipslack:~$
```

Si ext4 n'est pas présent, chargez le module **ext4.ko**. De plus, le noyau doit être compilé avec l'option **CONFIG_LSF** si vous souhaitez monter les systèmes de fichiers ext4 en lecture-écriture (ce qui doit toujours être le cas).

Les noyaux dont la version est inférieure à 2.6.28 disposent d'un module appelé **ext4dev.ko**. Le module **2.6.28 ext4.ko** peut monter des partitions ext4dev s'il est compilé avec l'option adéquate **CONFIG_EXT4DEV_COMPAT**.

3.1 Les outils à mettre à jour

Les outils manipulant le système de fichiers sont généralement regroupés en un paquet appelé **e2fsprogs**. Seules les dernières versions de distributions disposent d'un paquet suffisamment à jour. Il est nécessaire d'utiliser les **e2fsprogs** en version 1.41.3 ou supérieurs.

La Debian stable utilise les **e2fsprogs** en version 1.39, ce qui est insuffisant. La Debian *testing* dispose du paquet à jour.

La Slackware 12.2 possède, elle aussi, un paquet suffisamment récent.

La Fedora 10 dispose des paquets à jour.

La commande suivante indique la version des outils sur une Slackware 12.1 :

```
kevin@zipslack:~$ uname -r
2.6.28
kevin@zipslack:~$ /sbin/tune2fs -v
tune2fs 1.40.8 (13-Mar-2008)
  (…snip…)
```

Les outils doivent donc être mis à jour sur une Slackware 12.1. Ils se téléchargent sur le wiki d'**e2fsprogs** : http://e2fsprogs.sourceforge.net/. La compilation répond au triptyque habituel :

```
kevin@zipslack:/usr/src/e2fsprogs-1.41.3$ ./configure && make
  (…snip…)
kevin@zipslack:/usr/src/e2fsprogs-1.41.3$ su
Mot de passe:
root@zipslack:/usr/src/e2fsprogs-1.41.3# make install
```

Le **./configure** va installer les programmes directement sous la racine, c'est-à-dire **/** au lieu d'**/usr/local**. Par conséquent, la commande **make install** va remplacer tous les **e2fsprogs** par la version 1.41.3 compatible ext4. Pour l'éviter, vous pouvez choisir d'installer les programmes dans **/usr/local/** avec la commande :

```
kevin@zipslack:/usr/src/e2fsprogs-1.41.3$ ./configure --prefix=/usr/
local && make
```

Avant d'utiliser ces outils, remplacez le fichier **/etc/mke2fs.conf** par celui fourni avec les sources. C'est **obligatoire** pour le bon fonctionnement des outils.

3.2 Sur un nouveau système de fichiers

La méthode la plus simple et la moins risquée pour utiliser ext4 consiste à l'employer sur un nouveau disque. L'emploi d'une image *loop* est tout à fait possible :

```
root@zipslack:/tmp/article# dd if=/dev/zero of=disk30M bs=1024k count=30
30+0 enregistrements lus
30+0 enregistrements écrits
31457280 bytes (31 MB) copied, 1,50321 s, 20,9 MB/s
root@zipslack:/tmp/article# mkfs.ext4 disk30M
mke2fs 1.41.3 (12-Oct-2008)
disk30M n'est pas un périphérique spécial en mode bloc.
Procéder malgré tout ? (o,n) o
Étiquette de système de fichiers=
Type de système d'exploitation : Linux
Taille de bloc=1024 (log=0)
Taille de fragment=1024 (log=0)
7680 i-noeuds, 30720 blocs
1536 blocs (5.00%) réservés pour le super utilisateur
Premier bloc de données=1
Nombre maximum de blocs du système de fichiers=31457280
4 groupes de blocs
8192 blocs par groupe, 8192 fragments par groupe
1920 i-noeuds par groupe
Superblocs de secours stockés sur les blocs :
  8193, 24577
Écriture des tables d'i-noeuds : complété
Creating journal (1024 blocks): complété
Écriture des superblocs et de l'information de comptabilité du système de
fichiers : complété
Le système de fichiers sera automatiquement vérifié tous les 39
montages ou après 180 jours, selon la première éventualité. Utiliser
tune2fs -c ou -i pour écraser la valeur.
root@zipslack:/tmp/article# mount -t ext4 -o loop disk30M /mnt/hd
root@zipslack:/tmp/article# mount | grep disk
/tmp/article/disk30M on /mnt/hd type ext4 (loop=/dev/loop0)
root@zipslack:/tmp/article#
```

3.3 Migrer des partitions existantes

Migrer une partition ext2 directement vers ext4 est impossible. Il faut impérativement migrer une partition ext2 vers ext3 préalablement à l'aide de **tune2fs** :

```
root@slackware:/# tune2fs -j /dev/vda2
tune2fs 1.41.3 (12-Oct-2008)
Creating journal inode: done
This filesystem will be automatically checked every 20 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
root@slackware:/#
```

Migrer une partition ext3 vers une partition ext4 est réalisable. Toutefois, il faut savoir qu'il s'agit d'une opération à sens unique. Une fois la partition migrée en ext4, il faudra toujours la monter en ext4. Les changements induits par ext4 sont trop importants pour pouvoir réutiliser ext3. Cette méthode diffère de la migration ext2 vers ext3, car le retour arrière était possible.

Un deuxième point important concerne Grub. En effet, aucune des versions de Grub installées par les distributions ne sait lire ext4. Si la partition contenant **/boot** est migrée en ext4, alors Grub ne saura plus trouver le noyau, donc la machine ne sera plus *bootable*. Plus précisément, les dernières versions svn de Grub savent lire de l'ext4 et sont donc compatibles ext4, mais les distributions ne sont pas à jour. Le code est considéré comme expérimental, donc, pour l'instant, il est déconseillé de migrer **/boot** vers ext4.

Lilo a le même problème.

La migration d'une partition s'effectue en plusieurs étapes. Tout d'abord, il faut étendre les propriétés du système de fichiers à l'aide de l'outil **tune2fs**, puis le corriger à l'aide de **fsck** pour enfin pouvoir le monter. Bien entendu, toutes ces opérations doivent être réalisées alors que le disque est non monté.

L'exemple qui suit consiste à effectuer la manipulation sur un disque loop formaté en ext3, rempli avec quelques fichiers.

```
root@zipslack:/tmp/article# dd if=/dev/zero of=ext3-ext4.img bs=1024k count=30
30+0 enregistrements lus
30+0 enregistrements écrits
31457280 bytes (31 MB) copied, 2,51661 s, 12,5 MB/s
root@zipslack:/tmp/article# mkfs.ext3 ext3-ext4.img
mke2fs 1.41.3 (12-Oct-2008)
  (…snip…)
root@zipslack:/tmp/article# mount -t ext3 -o loop ext3-ext4.img /mnt/hd/
root@zipslack:/tmp/article# cp /boot/bzImage-2.6.28 /mnt/hd/
root@zipslack:/tmp/article# echo "Welcome to ext4" > /mnt/hd/texte
root@zipslack:/tmp/article# mkdir -p /mnt/hd/a/b/c/d
root@zipslack:/tmp/article# umount /mnt/hd/
root@zipslack:/tmp/article#
```

La migration commence ; elle se passe en deux temps. Tout d'abord, l'ajout des paramètres propres à ext4 avec **tune2fs**, puis la transformation à l'aide de **fsck** du système ext3 vers un système ext4. Le **fsck** est nécessaire et obligatoire. Sans celui-ci, le montage du disque ne se fera pas :

```
root@zipslack:/tmp/article# tune2fs -O extents,uninit_bg,dir_index
ext3-ext4.img
tune2fs 1.41.3 (12-Oct-2008)
root@zipslack:/tmp/article# fsck.ext4 -pf ext3-ext4.img
ext3-ext4.img: Groupe descriptor 0 checksum is invalid.  CORRIGÉ.
ext3-ext4.img: Groupe descriptor 1 checksum is invalid.  CORRIGÉ.
ext3-ext4.img: Groupe descriptor 2 checksum is invalid.  CORRIGÉ.
ext3-ext4.img: Groupe descriptor 3 checksum is invalid.  CORRIGÉ.
ext3-ext4.img: 17/7680 files (0.0% non-contiguous), 5067/30720 blocks
root@zipslack:/tmp/article# mount -t ext4 -o loop ext3-ext4.img /mnt/hd/
root@zipslack:/tmp/article# ls /mnt/hd/
a/  bzImage-2.6.28  lost+found/  texte
root@zipslack:/tmp/article# cat /mnt/hd/texte
Welcome to ext4
root@zipslack:/tmp/article#
```

Les erreurs sont tout à fait normales. **fsck.ext4** transforme le système de fichiers ext3 en système ext4. C'est également pour cette raison qu'il est impossible par la suite de redescendre en ext3.

3.4 Migration de la partition racine

La migration d'une distribution se fera de la même manière. L'utilisation de Lguest (un virtualiseur) comme plateforme de développement permet de tester sans risque la migration. De plus, Lguest permet de lancer directement un noyau, évitant ainsi de mettre à jour Grub ou Lilo.

Il suffit de booter en mode *single*, de changer ext3 par ext4 dans le **fstab** pour les partitions concernées, de lancer les actions de migration, puis de rebooter sur le nouveau système, une Slackware 12.2 installée pour l'occasion :

```
root@(none):~# vi /etc/fstab
  (changement d'ext3 en ext4)
root@(none):~# mount -o remount,ro /
root@(none):~# tune2fs -O extents,uninit_bg,dir_index /dev/vda2
tune2fs 1.41.3 (12-Oct-2008)
Please run e2fsck on the filesystem.
(and reboot afterwards!)
root@(none):~# fsck.ext4 -pf /dev/vda2
/dev/vda2: Group descriptor 0 checksum is invalid.  FIXED.
/dev/vda2: Group descriptor 1 checksum is invalid.  FIXED.
/dev/vda2: Group descriptor 2 checksum is invalid.  FIXED.
/dev/vda2: Group descriptor 3 checksum is invalid.  FIXED.
/dev/vda2: Group descriptor 4 checksum is invalid.  FIXED.
/dev/vda2: Group descriptor 5 checksum is invalid.  FIXED.
/dev/vda2: Group descriptor 6 checksum is invalid.  FIXED.
/dev/vda2: ***** REBOOT LINUX *****
/dev/vda2: 25110/57232 files (0.2% non-contiguous), 75148/228926 blocks
root@(none):~# halt
```

Le système de fichiers racine est passé en ext4. Un *reboot* va le confirmer :

```
 (…snip…)
IO APIC resources could be not be allocated.
Using IPI Shortcut mode
EXT3-fs: vda2: couldn't mount because of unsupported optional features (40).
EXT4-fs: barriers enabled
kjournald2 starting.  Commit interval 5 seconds
EXT4-fs: delayed allocation enabled
EXT4-fs: file extents enabled
EXT4-fs: mballoc enabled
EXT4-fs: mounted filesystem with ordered data mode.
VFS: Mounted root (ext4 filesystem) readonly.
Freeing unused kernel memory: 296k freed
INIT: version 2.86 booting
 (…snip…)
Welcome to Linux 2.6.28 (/dev/hvc0)
lguest login: root
Password:
Linux 2.6.28.
Last login: Mon Jan 12 18:20:24 +0000 2009 on /dev/hvc0.
You have mail.
root@lguest:~# mount
/dev/root on / type ext4 (rw,barrier=1,data=ordered)
/proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
root@lguest:~#
```

La distribution Slackware 12.2 est fonctionnelle sur une racine en ext4.

4 Il n'y aura pas d'ext5

Après ext4, ext5 ? La question du système de fichiers suivant est légitime. Il est déjà possible de dire qu'il n'y aura pas un système de fichiers ext5.

Remy Card le développeur d'ext2 a prévu son système de fichiers de façon à ce qu'il puisse être amélioré. Toutefois, après plus de 15 ans d'âge (il a été écrit en 1993), il semble avoir touché ses limites. Le développeur d'ext3 et ext4 (Theodore Ts'o) et un groupe de développeurs de systèmes de fichiers ont conclu qu'il fallait repenser entièrement un nouveau système de fichiers. http://thread.gmane.org/gmane.linux.file-systems/26246/focus=26492.

Le prochain système de fichiers sous Linux sera btrfs (le lire *Better FS* ou Meilleur système de fichiers). Il est encore en version bêta et n'intégrera le noyau Linux qu'avec le 2.6.29 en mode **-dev**, donc à ne pas utiliser en production.

Theodore Ts'o a donc appelé ext4 le « chaînon manquant » entre les systèmes de fichiers ext et btrfs.

btrfs semble très prometteur, puisque capitalisant sur l'ensemble des systèmes de fichiers, à commencer par la série des ext. Lire : http://btrfs.wiki.kernel.org/.

5 Conclusion

Une simple question : faut il migrer sous ext4 ? Certainement. Les améliorations d'ext4 sont bénéfiques pour tout usage des systèmes de fichiers, certains tests montrant une machine plus rapide sans autre différence qu'ext4.

Certaines distributions proposent déjà ext4 dès l'installation. La migration des partitions étant possible, il n'existe pas de raison de rester sous ext3.

Auteur : Kevin Denis

ZFS sous GNU/Linux

Un système de fichiers révolutionnaire qui s'auto-répare, permet l'utilisation d'un genre de raid (niveaux 0, 1, 5 et même 6), permet la compression et la création de snapshots à la volée, possède des limites qui n'ont aucune signification physique aujourd'hui et fonctionne sous GNU/Linux. Pas possible ? ZFS !

Auteur

- Olivier Delhomme

ZFS est un système de fichiers dédié serveur qui promet de réconcilier les administrateurs système avec les systèmes de fichiers. Il a été écrit par les développeurs de Sun. Il est donc disponible nativement sous Solaris. Il simplifie toutes les opérations qui nous horripilent ou nous rebutent. Il a une fonctionnalité qui m'a vraiment fait craquer et qui me paraît tellement évidente qu'on se demande pourquoi les autres systèmes de fichiers ne la proposent pas : il est capable (avec un système de disques redondants) de s'auto-réparer ! Le code **[0]** est placé sous licence CDDL, approuvée par l'OSI (l'*Open Source Initiative* **[1]**), c'est donc un logiciel libre même si l'on peut regretter ce choix pour des raisons éthiques **[2]**. Voyons ce dont ZFS est capable.

1 Des chiffres

Lorsque l'on parle de chiffres, pour un système de fichiers, il s'agit généralement de ses limites. ZFS, dans ce domaine, a été conçu directement pour le futur, jugez plutôt :

- Système de fichiers entièrement 128 bits, c'est-à-dire non seulement pour le nombre d'*inodes*, mais aussi pour la taille maximale d'un fichier .
- 2^{64} soit plus de 18 milliards de milliards de fichiers au maximum dans chaque système de fichiers.
- 2^{48} *snapshots* au maximum. Si on en fait 10 par seconde, on peut tenir pendant plus de 890 000 ans ! Mais, il faudrait de la place et justement...
- 16 exa octets, soit plus de 16 000 peta octets. C'est la taille maximale d'un système de fichiers. 16 exa octets, c'est aussi la taille maximale d'un fichier !
- 2^{48} fichiers au maximum dans un répertoire.
- 256 zeta octets, soit plus de 256 000 exa octets. C'est la taille maximale d'un pool (un ensemble de périphériques) qui peut contenir au maximum 2^{64} systèmes de fichiers.
- Les pools peuvent être formés d'au maximum 2^{64} périphériques.

Ces chiffres sont tellement immenses qu'on ne se rend pas bien compte de ce que cela représente. Aussi, il est probable que l'énergie nécessaire pour remplir un système de fichiers comme celui-là soit si gigantesque qu'elle pourrait suffire à faire bouillir l'eau des océans **[3]**. On comprend mieux pourquoi ce système de fichiers a été nommé *Zeta File System*, soit *ZFS*.

2 Installation

Donc, en cherchant bien, on trouve Ricardo Correia (connu sous le pseudonyme wizeman), un gentil développeur, qui a porté ZFS pour qu'il utilise `fuse` sous GNU/Linux. En effet, la licence CDDL est incompatible avec la licence GPL (dans sa version 2), celle du noyau GNU/Linux. C'est pourquoi ZFS n'est pas intégré au noyau et que l'on doit utiliser `fuse` pour le faire fonctionner sous GNU/Linux. Ce n'est par exemple pas le cas de FreeBSD où ZFS est intégré nativement depuis la version 7.0.

Sur le site de FreeBSD **[4]**, il est écrit qu'il s'agit d'un support expérimental, mais un certain D.B., rédacteur en chef d'un magazine sur GNU/Linux me souffle qu'en réalité c'est « intégré de manière très très propre » !

Du coup, avant toute chose sous GNU/Linux (les noms des paquets cités ici proviennent de ma Debian), il faut installer `fuse-utils` et aussi `libfuse-dev`, car nous allons compiler le programme qui utilise les structures de `fuse`. Il faut aussi installer `libaio-dev` et `libaio1`

qui permettent d'effectuer les entrées/sorties de manière asynchrone. La bibliothèque de compression de données **zlib** vous sera demandée avec sa version de développement. Il faut également un noyau *GNU/Linux* 2.6.x supérieur à 2.6.15 et une **glibc** avec les *pthreads* dans une version supérieure à 2.3.3. Le système de compilation utilise la commande **scons** **[5]** en remplacement de la commande **make**. Il vous la faudra également.

Téléchargez **[6]** l'archive et décompressez-la avant de lancer la compilation :

```
$ tar jxvf zfs-fuse-0.5.0.tar.bz2
$ cd zfs-fuse-0.5.0/src/
$ scons
```

Si, comme moi, vous avez une machine dont l'âge est honorable, il faudra une bonne dizaine de minutes avant que cette opération ne se termine.

Installons le programme :

```
$ sudo scons install
```

Et voilà !!

Par défaut, tout s'installe dans **/usr/local/sbin**. Vous pouvez modifier cela en spécifiant une option à **scons** :

```
$ sudo scons install install_dir=/chemin/vers/les/programmes
```

3 Tour d'horizon rapide des commandes et des fonctionnalités

Nous venons d'installer les commandes **zdb**, **zfs**, **zfs-fuse**, **zpool** et **ztest**.

zdb est la commande qui permet de diagnostiquer des pannes et erreurs de ZFS. Toutefois, comme le fonctionnement de ZFS lui garantit d'être toujours cohérent et qu'il se répare tout seul, cette commande n'est pas très utile dans un premier temps.

zpool permet de gérer les « pools » de stockage, c'est-à-dire les regroupements de disques de stockage.

zfs permet de configurer le ou les systèmes de fichiers contenus dans les « pools ».

ztest permet de faire des tests.

zfs-fuse va nous servir sous GNU/Linux puisqu'il s'agit du mauvais génie (le démon quoi !) qui gère tout ça !!

Mentionnons l'option pour obtenir l'aide des commandes. Cette dernière n'étant pas très POSIX **[7]**, elle est un peu inhabituelle. Aussi, vous devrez utiliser l'option **-?**.

Les fonctionnalités présentes dans la version 0.5 sont les suivantes :

- Créer et détruire des pools, des snapshots et des clones.
- Fonctionnalités de raid, raid-0 ou agrégats, raid 1 ou miroir, raid 5 ou à parité contrôlée (cette fonctionnalité se nomme raidz pour ZFS), raid 6 à double parité (nommée raidz2).
- Possibilité d'utiliser n'importe quel périphérique en mode bloc ou n'importe quel fichier (sauf un issu d'un système de fichiers ZFS) pour réaliser un périphérique virtuel.
- La compression, la détection d'erreur, la vérification des données ainsi que l'auto-réparation (sur un volume de type raid redondant type raid 1, raid 5 ou raid 6).
- Les quotas et la réservation fonctionnent, même s'il ne s'agit pas de « vrais quotas » à la sauce POSIX du terme.
- Sauvegarde et restauration (via les snapshots).

Bien qu'il y ait déjà de nombreuses fonctionnalités opérationnelles, il en existe beaucoup d'autres qui ne le sont pas encore, soit qu'elles ne sont pas encore implémentées, soit qu'elles ne peuvent l'être du fait de limitations dues principalement à **fuse** (que l'on remercie quand même d'exister). Pour avoir une idée plus précise des fonctionnalités qui restent à écrire, reportez-vous au fichier STATUS qui se trouve à la racine des sources.

4 Passons aux choses sérieuses

4.1 Création d'un pool

Avant toute chose, vérifiez que **fuse** est bien là et lancer le processus ZFS pour ce dernier comme suit. Attention, la commande ne se termine pas, ouvrez donc un terminal dédié :

```
$ sudo -s
# modprobe fuse
# zfs-fuse --no-daemon
```

Si cela ne fonctionne pas, reportez vous au paragraphe 6. Pour faire mes tests, vu qu'à la maison je n'ai pas un serveur de fichiers avec plusieurs contrôleurs et plusieurs disques, j'utilise des fichiers comme s'ils étaient des disques. Je me crée dix « disques » de 512 Mo :

```
$ mkdir test_zfs
$ cd test_zfs
$ for i in $(seq 0 9); \
do dd if=/dev/zero of=./disque_$i bs=1M count=512; \
done;
```

Après un long moment (ma machine a mis plus de 5 minutes), vous obtenez dix magnifiques fichiers, tous d'une taille identique de 512 Mo. Nous admettrons qu'il s'agit là de dix disques durs très haute performance !

Attention

La création d'un pool nécessite des périphériques (ou fichiers) dont la taille est supérieure ou égale à 64 Mo.

Donc, nous allons créer un pool avec les dix disques, en raidz2, c'est-à-dire un genre de raid 6 et disons que, comme on est très prudent, on va utiliser un disque de réserve (« *spare* » en anglais). On souhaite aussi que le volume correspondant soit monté sur **/media/zfs_test** (qui est un dossier qui n'existe pas sur le système utilisé). Utilisons la commande suivante :

```
# zpool create raid6 raidz2 \
/tmp/test_zfs/disque_{0,1,2,3,4,5,6,7,8} \
spare /tmp/test_zfs/disque_9 -m /media/zfs_test
```

Note

S'agissant de fichiers et non de périphériques, veillez à bien spécifier le chemin absolu (depuis la racine) de chaque fichier.

Hop et voilà !! Si si, c'est tout ce qu'il y a à faire. D'ailleurs, vérifions tout de suite avec la commande **df -h** qui donne :

```
# df -h
Sys. de fich.        Tail. Occ. Disp. %Occ. Monté sur
/dev/hda2             12G  7,9G  3,5G  70% /
tmpfs                237M     0  237M   0% /lib/init/rw
udev                  10M  104K  9,9M   2% /dev
tmpfs                237M     0  237M   0% /dev/shm
/dev/hda1            236M  145M   79M  65% /boot
/dev/hda5             24G  9,2G   14G  41% /home
raid6                3,4G   42K  3,4G   1% /media/zfs_test
```

3,4 Go de libre, c'est bien 7*512Mo, car, pour schématiser, on a 9 disques dans le raidz2 dont *2* disques de parités soit 7 disques utiles plus un disque de réserve (ici spare). Cette place est directement utilisable, car montée automatiquement sur **/media/zfs_test** (comme on lui avait demandé). Ici, pas besoin d'un long **mkfs**. La commande s'est exécutée en un clin d'œil : la taper a presque été plus long ! Une commande utile à ce stade est **zpool status** qui permet de connaître l'état des pools (ici un seul) :

```
# zpool status
 pool: raid6
 state: ONLINE
 scrub: none requested
config:

        NAME                    STATE     READ WRITE CKSUM
        raid6                   ONLINE       0     0     0
          raidz2                ONLINE       0     0     0
            /tmp/zfs_test/disque_0  ONLINE       0     0     0
            /tmp/zfs_test/disque_1  ONLINE       0     0     0
            /tmp/zfs_test/disque_2  ONLINE       0     0     0
            /tmp/zfs_test/disque_3  ONLINE       0     0     0
            /tmp/zfs_test/disque_4  ONLINE       0     0     0
            /tmp/zfs_test/disque_5  ONLINE       0     0     0
            /tmp/zfs_test/disque_6  ONLINE       0     0     0
            /tmp/zfs_test/disque_7  ONLINE       0     0     0
            /tmp/zfs_test/disque_8  ONLINE       0     0     0
        spares
          /tmp/zfs_test/disque_9    AVAIL

errors: No known data errors
```

Ici, on voit très nettement qu'il y a 9 disques dans le raidz2 que j'ai nommé « raid6 » et un disque de réserve qui est disponible (*AVAIL*). Cette commande nous servira, par la suite, pour connaître l'état du pool.

Copions donc des données. Une commande **du -hs /var/cache** m'indique que ce dernier occupe 3,4 Go, parfait pour être copié dans mon volume de test :

```
# cp -a /var/* /media/zfs_test/
```

Là, pour le coup, c'est long, très long même, plus de 30 minutes sur ma machine. Mais c'est normal, car toutes les opérations ont lieu sur le même disque physique. De plus, ZFS crée le système de fichiers à la volée et calcule les sommes de contrôle (*checksums*) qui lui seront utiles au cas où il lui arriverait malheur (et croyez-moi, je ne vais pas être tendre). Si, comme moi, vous essayez de mettre plus de données que le volume ne peut en contenir, vous allez vous rendre compte, vers la fin, que ZFS cherche de la place et se réorganise. Si vous regardez la place libre avec la commande **df -h** régulièrement, il est possible que vous voyiez cette dernière augmenter, alors même que vous copiez des données !! Astuce : Pour réaliser cela, utilisez la commande **watch** comme suit (utilisez [Ctrl-C] pour quitter le programme) :

```
$ watch -t -d -n1 "df -h"
```

La recherche de place libre est très consommatrice de ressources et écroule totalement les performances (sous GNU/Linux en tout cas). Il est préférable de ne jamais remplir totalement un système de fichiers ZFS pour éviter ce problème. Notamment, cela vous évitera de vous demander pourquoi vous n'arrivez pas à effacer un fichier de 2 Go quand votre pool ZFS est plein à 99%...

Notons que ZFS gère plusieurs types de systèmes de fichiers tels que « raidz2 » que nous venons de voir ou « raidz » qui est une sorte de raid5 et « mirror » qui est une sorte de raid1. De base, il gère le raid0 qui est un simple agrégat. Nous le verrons plus tard. Il est possible de mélanger, dans une certaine mesure, tout cela.

4.2 Faisons souffrir le système de fichiers

4.2.1 Un disque rempli d'erreurs

Écrivons n'importe quoi dans le fichier **disque_4** par exemple :

```
# shred -n 1 disque_4
```

Après une manœuvre pareille, le contenu du fichier **disque_4** n'est plus du tout le même qu'avant.

La commande **zpool status** annonce qu'il y a eue une erreur, mais que les applications n'ont pas été affectées. Elle indique aussi les actions à réaliser pour effacer les erreurs ou pour remplacer le disque défectueux. Par contre, l'accès aux fichiers dans **/media/zfs_test/** reste possible et sans erreurs, car ZFS les détecte et les corrige à la volée. À noter que la commande **zpool scrub raid6** permet de corriger l'ensemble des données du volume (moins le volume est rempli, plus c'est rapide).

La colonne CKSUM de la commande **zpool status** indique le nombre de sommes de contrôle trouvées en erreur.

4.2.2 Un disque en moins

Bon, dans la vraie vie, c'est rare que l'on puisse réécrire sur les secteurs défectueux d'un disque. Pour simuler un disque défectueux, tuez le processus **zfs-fuse**, supprimez le fichier de votre choix, relancez le processus **zfs-fuse** et réimportez le pool (voir § 6.) :

```
# killall zfs-fuse
# rm -f disque_4
# zpool import -d . raid6
# zpool status

  pool: raid6
 state: DEGRADED
status: One or more devices could not be opened.  Sufficient replicas exist for
  the pool to continue functioning in a degraded state.
action: Attach the missing device and online it using 'zpool online'.
   see: http://www.sun.com/msg/ZFS-8000-2Q
 scrub: none requested
config:

  NAME                      STATE     READ WRITE CKSUM
  raid6                     DEGRADED     0     0     0
    raidz2                  DEGRADED     0     0     0
      /tmp/zfs_test/disque_0  ONLINE       0     0     0
      /tmp/zfs_test/disque_1  ONLINE       0     0     0
      /tmp/zfs_test/disque_2  ONLINE       0     0     0
      /tmp/zfs_test/disque_3  ONLINE       0     0     0
      7760834922797222775     UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_4
      /tmp/zfs_test/disque_5  ONLINE       0     0     0
      /tmp/zfs_test/disque_6  ONLINE       0     0     0
      /tmp/zfs_test/disque_7  ONLINE       0     0     0
      /tmp/zfs_test/disque_8  ONLINE       0     0     0
  spares
    /tmp/zfs_test/disque_9    AVAIL

errors: No known data errors
```

L'indication est claire : il manque un disque. La commande conseille de le brancher et de le mettre en ligne en utilisant la commande **zpool online**.

4.2.3 Deux disques en moins ! Vite, utilisons le disque de réserve !

Soyons fous, recommençons et supprimons **disque_5**. Les sorties sont les mêmes, mais, cette fois, nous n'avons plus la ceinture, ni les bretelles et le moindre problème dans l'un des disques restant serait fatal aux données. Utilisons donc le disque de spare :

```
# zpool replace /tmp/zfs_test/disque_4 /tmp/zfs_test/disque_9
```

Et voilà ! Comment ça et voilà ? Si si, je vous assure, c'est tout, ZFS s'occupe de reconstruire en arrière-plan le système de fichiers, de noter que le périphérique **disque_9** est maintenant utilisé et qu'il remplace **disque_4** et vous continuez d'avoir accès à vos fichiers de manière transparente, comme si de rien était :

```
# zpool status
  pool: raid6
 state: DEGRADED
status: One or more devices could not be opened.  Sufficient replicas exist for
  the pool to continue functioning in a degraded state.
action: Attach the missing device and online it using 'zpool online'.
   see: http://www.sun.com/msg/ZFS-8000-2Q
 scrub: resilver in progress for 0h2m, 11,20% done, 0h16m to go
config:

  NAME                      STATE     READ WRITE CKSUM
  raid6                     DEGRADED     0     0     0
    raidz2                  DEGRADED     0     0     0
      /tmp/zfs_test/disque_0  ONLINE       0     0     0
      /tmp/zfs_test/disque_1  ONLINE       0     0     0
      /tmp/zfs_test/disque_2  ONLINE       0     0     0
      /tmp/zfs_test/disque_3  ONLINE       0     0     0
      spare                   DEGRADED     0     0     0
        7760834922797222775   UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_4
        /tmp/zfs_test/disque_9  ONLINE       0     0     0
      2705405041636850921     UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_5
      /tmp/zfs_test/disque_6  ONLINE       0     0     0
      /tmp/zfs_test/disque_7  ONLINE       0     0     0
      /tmp/zfs_test/disque_8  ONLINE       0     0     0
  spares
    /tmp/zfs_test/disque_9    INUSE     currently in use

errors: No known data errors
```

Regardez bien. La commande indique même depuis combien de temps elle reconstruit (ici 2 minutes) et combien de temps il va lui falloir pour terminer (à vitesse constante, ici 16 minutes – en réalité, il ne lui a fallu que 12 minutes).

4.2.4 Allons-y gaiement...

Ce qui est agréable avec la commande **zpool status**, c'est qu'elle indique tout de manière précise et concise. Par exemple, supprimons le disque de spare (**disque_9**) et inscrivons 10 Mo de données aléatoires dans l'un des disques restants. Le résultat sera qu'un certain nombre de fichiers seront illisibles, oui, mais lesquels ?

```
# rm -f disque_9
# dd if=/dev/urandom of=disque_7 bs=1024 count=10000 \ seek=56654
conv=notrunc
10000+0 enregistrements lus
10000+0 enregistrements écrits
10240000 bytes (10MB) copied, 3,51317 s, 2,9 MB/s
```

```
# zpool import -d . raid6
# zpool status
  pool: raid6
 state: DEGRADED
status: One or more devices could not be opened.  Sufficient replicas exist for
  the pool to continue functioning in a degraded state.
action: Attach the missing device and online it using 'zpool online'.
   see: http://www.sun.com/msg/ZFS-8000-2Q
 scrub: none requested
config:

  NAME                      STATE     READ WRITE CKSUM
  raid6                     DEGRADED     0     0     0
    raidz2                  DEGRADED     0     0     0
      /tmp/zfs_test/disque_0  ONLINE       0     0     0
      /tmp/zfs_test/disque_1  ONLINE       0     0     0
      /tmp/zfs_test/disque_2  ONLINE       0     0     0
      /tmp/zfs_test/disque_3  ONLINE       0     0     0
      spare                   UNAVAIL      0     0     0  insufficient
replicas
        7760834922797222775   UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_4
        9623051411205814655   UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_9
      2705405041636850921     UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_5
      /tmp/zfs_test/disque_6  ONLINE       0     0     0
      /tmp/zfs_test/disque_7  ONLINE       0     0     0
      /tmp/zfs_test/disque_8  ONLINE       0     0     0
  spares
    /tmp/zfs_test/disque_9    UNAVAIL   cannot open

errors: No known data errors
# zpool scrub raid6
#
```

Après l'import du nouveau pool très dégradé, **zpool status** nous indique bien avoir perdu le disque de réserve et qu'il n'y a pas assez de disques pour la réserve. Mais, la commande n'indique pas les éventuelles erreurs dans le système de fichiers (erreurs dues à la commande **dd**). La commande **zpool scrub raid6** permet de parcourir l'ensemble des données pour y détecter des erreurs et éventuellement les corriger, le tout de manière transparente. Voyons le résultat avec la commande **zpool status -v** :

```
# zpool status -v
  pool: raid6
 state: DEGRADED
status: One or more devices has experienced an error resulting in data
  corruption.  Applications may be affected.
action: Restore the file in question if possible.  Otherwise restore the
  entire pool from backup.
   see: http://www.sun.com/msg/ZFS-8000-8A
 scrub: scrub in progress for 0h5m, 77,37% done, 0h1m to go
config:

  NAME                      STATE     READ WRITE CKSUM
  raid6                     DEGRADED 1,07K     0     0
    raidz2                  DEGRADED 1,07K     0     0
      /tmp/zfs_test/disque_0  ONLINE       0     0     0
      /tmp/zfs_test/disque_1  ONLINE       0     0     0
      /tmp/zfs_test/disque_2  ONLINE       0     0     0
      /tmp/zfs_test/disque_3  ONLINE       0     0     0
      spare                   UNAVAIL      0     0     0  insufficient
replicas
        7760834922797222775   UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_4
        9623051411205814655   UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_9
      2705405041636850921     UNAVAIL      0     0     0  was /tmp/
zfs_test/disque_5
      /tmp/zfs_test/disque_6  ONLINE       0     0     0
      /tmp/zfs_test/disque_7  ONLINE       0     0    39
      /tmp/zfs_test/disque_8  ONLINE       0     0     0
  spares
    /tmp/zfs_test/disque_9    UNAVAIL   cannot open

errors: Permanent errors have been detected in the following files:

        /media/zfs_test/apt/archives/iceape-browser_1.1.9-5_i386.deb
        /media/zfs_test/apt/archives/libnspr4-0d_4.7.1-3_i386.deb
        /media/zfs_test/apt/archives/libedataserver1.2-9_2.22.2-1_i386.deb
        /media/zfs_test/apt/archives/libcamel1.2-11_2.22.2-1_i386.deb
        /media/zfs_test/apt/archives/libexchange-storage1.2-3_2.22.2-1_i386.deb
        /media/zfs_test/apt/archives/libgnome-pilot2_2.0.15-2.4_i386.deb
        /media/zfs_test/apt/archives/libgtkhtml3.14-19_3.18.2-1_i386.deb
        /media/zfs_test/apt/archives/libnm-glib0_0.6.6-1_i386.deb
        /media/zfs_test/apt/archives/libpisock9_0.12.3-5_i386.deb
        /media/zfs_test/apt/archives/libpisync1_0.12.3-5_i386.deb
        /media/zfs_test/apt/archives/libpanel-applet2-0_2.20.3-5_i386.deb
        /media/zfs_test/apt/archives/evolution-common_2.22.2-1.1_all.deb
        /media/zfs_test/apt/archives/libwnck22_2.22.3-1_i386.deb
        /media/zfs_test/apt/archives/libxslt1.1_1.1.24-1_i386.deb
```

Et ainsi de suite... La commande indique non seulement le disque où se sont produites les erreurs, via la colonne CKSUM, mais aussi très précisément quels sont les fichiers qui ne sont plus bons. Dans l'exemple, il s'agit de 79 fichiers du dossier **/apt/archives/**.

Dans le cas d'un système de fichiers réel, si une telle chose se produit, l'administrateur connaît rapidement le nom des fichiers erronés et peut, après avoir réparé, c'est-à-dire remplacé les disques, faire une restauration à partir de la sauvegarde.

4.2.5 Remplacement des disques

Comme vous avez des données très importantes sur vos disques, vous commandez de nouveaux disques qui iront remplacer les anciens. Manque de bol, la capacité ridicule de 512 Mo n'existe plus et les seuls nouveaux disques que vous trouvez sont des disques de 650 Mo :

```
# dd if=/dev/zero of=nouveau_0 bs=1M count=650
# dd if=/dev/zero of=nouveau_1 bs=1M count=650
```

Comment va se comporter le système avec des disques de taille supérieure ? Est-ce qu'il va subsister de la place non utilisée sur le disque ?

Le remplacement des disques manquant est possible, même avec des disques de plus grande taille. Toutefois, il n'est pas complètement effectif dans notre exemple, notamment, le système refuse obstinément d'enlever l'ancien disque arguant du fait qu'il manque des données. Supprimons donc ces données qui sont de toute manière perdues (on pensera à remercier l'administrateur d'avoir fait des sauvegardes sur bande) :

```
# zpool status -v raid6 | grep media | xargs rm -f
```

Demandons-lui de se réparer avec une commande du type **fsck** :

```
# zpool scrub raid6
```

La commande s'exécute en arrière-plan et **zpool status raid6** vous dit où elle en est. En attendant qu'elle termine, profitez-en pour regarder le résultat de la commande **zpool iostat -v raid6**. Elle indique la place utilisée et restante des pools et systèmes de fichiers, ainsi que le nombre d'opérations d'écriture et de lecture par secondes effectuées sur chaque disque et la bande passante que cela représente. Il est possible de demander à ce que l'exécution de cette commande se répète selon un intervalle donné, par exemple toutes les 3 secondes : **zpool iostat -v raid6 3**.

Lorsque la commande est terminée, vous pouvez enlever les anciens disques défectueux avec la commande **zpool detach raid6 /tmp/zfs_test/disque_5** ou en utilisant le nom système du disque (dans notre exemple, il s'agit de : **2705405041636850921**).

La commande **zpool list** qui donne un résumé très synthétique des pools indique qu'il n'y a pas eu d'augmentation de la capacité. Si l'on souhaite que la capacité augmente, il faudra remplacer tous les anciens disques avec des nouveaux de capacité identique. Tout cela se réalise bien entendu alors que le système de fichiers est monté. Vous pouvez accéder à vos fichiers en toute transparence !

Un bémol toutefois, pour que mon système prenne effectivement en compte l'augmentation de la capacité totale, il a fallu que j'arrête puis relance **zfs-fuse** et que j'importe à nouveau le pool. Cette manipulation n'a pris que quelques secondes, mais, pendant ces quelques secondes, les fichiers ne sont plus accessibles (pensez à prévenir les utilisateurs).

4.2.6 Augmentation de capacité par ajout d'un nouveau raidz au pool

Un autre moyen d'augmenter la capacité est d'ajouter un nouvel ensemble de disques dans notre pool (que nous avons nommé raid6). Nous pouvons faire cela avec la commande suivante :

```
# zpool add raid6 raidz2 /tmp/zfs_test/dvd_{0,1,2,3,4,5,6,7,8}
```

Il faut garder en tête que ZFS a besoin d'un nombre de périphériques identiques entre ensemble de fichiers d'un même pool. Cela lui simplifie le travail (il est toutefois possible de le forcer avec l'option **-f**). Nous obtenons ainsi une sorte de raid 0+6 :

```
# zpool status -v
  pool: raid6
 state: ONLINE
 scrub: resilver completed after 0h11m with 0 errors on Thu Oct 16 22:35:33 2008
config:

        NAME                        STATE     READ WRITE CKSUM
        raid6                       ONLINE       0     0     0
          raidz2                    ONLINE       0     0     0
            /tmp/zfs_test/nouveau_3 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_4 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_2 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_5 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_0 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_1 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_6 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_7 ONLINE       0     0     0
            /tmp/zfs_test/nouveau_8 ONLINE       0     0     0
          raidz2                    ONLINE       0     0     0
            /tmp/zfs_test/dvd_0     ONLINE       0     0     0
            /tmp/zfs_test/dvd_1     ONLINE       0     0     0
            /tmp/zfs_test/dvd_2     ONLINE       0     0     0
            /tmp/zfs_test/dvd_3     ONLINE       0     0     0
            /tmp/zfs_test/dvd_4     ONLINE       0     0     0
            /tmp/zfs_test/dvd_5     ONLINE       0     0     0
            /tmp/zfs_test/dvd_6     ONLINE       0     0     0
            /tmp/zfs_test/dvd_7     ONLINE       0     0     0
            /tmp/zfs_test/dvd_8     ONLINE       0     0     0

errors: No known data errors
```

Soit une capacité portée à

```
# df -h
Sys. de fich.         Tail. Occ. Disp. %Occ. Monté sur
/dev/hda1             9,2G  5,1G  3,7G  59% /
udev                   10M   52K   10M   1% /dev
/dev/hda3             141G  102G   32G  77% /home
overflow              1,0M   12K 1012K   2% /tmp
raid6                  32G  1,8G   30G   6% /media/zfs_test
```

ou encore :

```
# zpool list
NAME    SIZE   USED  AVAIL    CAP  HEALTH  ALTROOT
raid6  40,7G  2,21G  38,4G     5%  ONLINE  -
```

En faisant ainsi, on a augmenté, de manière transparente, la taille totale du système de fichiers.

La taille donnée par la commande **zfs-list** et celle donnée par la commande **du -h** sont différentes. Il semble que la commande **zfs-list** ne prenne pas en compte dans son calcul les disques réservés au calcul des sommes de contrôle (deux dans notre cas).

4.2.7 Rappel des commandes utiles

zpool create nom_du_pool type_du_pool noms_des_périphériques crée un pool selon les options spécifiées. Le type du pool peut-être « mirror », « raidz » ou « raidz2 ».

zpool destroy nom_du_pool pour détruire un pool entier. Attention, cette commande ne demande pas de confirmation et il ne semble pas exister d'option « garde-fou » du type « **-i** » des *coreutils* !

zpool iostat -v permet l'affichage des statistiques d'entrées/sorties des pools tout en indiquant la répartition des données et de ces entrées/sorties sur chacun des fichiers ou périphériques constituant le pool.

zpool status -v donne l'état de chacun des pools. Cette commande détaille les erreurs lorsqu'il y en a.

5 Systèmes de fichiers

La commande zpool que nous venons de voir crée un système de fichiers racine dans le pool et le monte sur le point de montage qu'on lui a indiqué. Il est toutefois possible de créer, dans le pool, des systèmes de fichiers indépendants. Listons les systèmes de fichiers déjà utilisables :

```
# zfs list
NAME    USED  AVAIL  REFER  MOUNTPOINT
raid6  32,7M  26,6G  32,6M  /media/zfs_test
```

Nous retrouvons notre système de fichiers raid6 (qui est aussi le nom du pool) monté sur **/media/zfs_test**. Nous pouvons maintenant créer au maximum 264 systèmes de fichiers dans ce pool, par exemple :

```
# zfs create raid6/usr
# zfs create raid6/home
# zfs create raid6/var
```

Ce qui donne :

```
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
raid6       32,9M  26,7G  32,6M  /media/zfs_test
raid6/home  41,9K  26,7G  41,9K  /media/zfs_test/home
raid6/usr   41,9K  26,7G  41,9K  /media/zfs_test/usr
raid6/var   41,9K  26,7G  41,9K  /media/zfs_test/var
```

Vous pouvez vérifier avec la commande **df -h** que les systèmes de fichiers sont bien montés. Ils se partagent tous un même espace de stockage qui est le pool en lui-même. Si on le souhaite, on peut remplir **raid6/home** et, dans ce cas, il ne restera plus de place pour les autres systèmes de fichiers. On peut voir cela comme des répertoires. À la différence prêt qu'il est possible de préciser quelques options nommées « propriétés ». Par exemple, il est possible de redéfinir le point de montage :

```
# zfs set mountpoint=/media/sauvegarde raid6
# zfs set mountpoint=/media/var raid6/var
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
raid6       32,9M  26,7G  32,6M  /media/sauvegarde
raid6/home  41,9K  26,7G  41,9K  /media/sauvegarde/home
raid6/usr   41,9K  26,7G  41,9K  /media/sauvegarde/usr
raid6/var   41,9K  26,7G  41,9K  /media/var
# zfs mount raid6/var
```

La dernière commande permet de remonter le système de fichiers dont on vient de modifier le point de montage, le changement l'ayant démonté automatiquement. Si on change plusieurs points de montage, on pourra utiliser la commande **zfs mount -a**.

Pour obtenir les propriétés d'un système de fichiers, utilisez la commande **zfs get all nom_du_systeme_de_fichiers**, par exemple :

```
# zfs get all raid6/var
NAME      PROPERTY              VALUE                  SOURCE
raid6/var  type                 filesystem             -
raid6/var  creation             lun oct 27 23:10 2008  -
raid6/var  used                 41,9K                  -
raid6/var  available            26,7G                  -
raid6/var  referenced           41,9K                  -
raid6/var  compressratio        1.00x                  -
raid6/var  mounted              yes                    -
raid6/var  quota                none                   default
raid6/var  reservation          none                   default
raid6/var  recordsize           128K                   default
raid6/var  mountpoint           /media/var             local
raid6/var  sharenfs             off                    default
raid6/var  checksum             on                     default
raid6/var  compression          off                    default
raid6/var  atime                on                     default
raid6/var  devices              on                     default
raid6/var  exec                 on                     default
raid6/var  setuid               on                     default
raid6/var  readonly             off                    default
raid6/var  zoned                off                    default
raid6/var  snapdir              hidden                 default
raid6/var  aclmode              groupmask              default
raid6/var  aclinherit           restricted             default
raid6/var  canmount             on                     default
raid6/var  shareiscsi           off                    default
raid6/var  xattr                on                     default
raid6/var  copies               1                      default
raid6/var  version              3                      -
raid6/var  utf8only             off                    -
raid6/var  normalization        none                   -
raid6/var  casesensitivity      sensitive              -
raid6/var  vscan                off                    default
raid6/var  nbmand               off                    default
raid6/var  sharesmb             off                    default
raid6/var  refquota             none                   default
raid6/var  refreservation       none                   default
raid6/var  primarycache         all                    default
raid6/var  secondarycache       all                    default
raid6/var  usedbysnapshots      0                      -
raid6/var  usedbydataset        41,9K                  -
raid6/var  usedbychildren       0                      -
raid6/var  usedbyrefreservation 0                      -
```

Le paramètre **SOURCE** peut prendre trois valeurs :

- **default** qui indique une valeur par défaut (ou inchangée) ;
- **local** qui indique que la valeur a été fixée spécifiquement pour ce système de fichiers ;
- **inherited** qui indique que la valeur est héritée d'un système de fichiers parent.

On a déjà vu que l'on peut changer le point de montage en changeant la propriété **mountpoint**, mais ce changement nécessite de remonter les systèmes de fichiers. Heureusement, ce n'est pas toujours le cas et, notamment, il est possible de changer, de manière transparente et alors que le système de fichiers est en fonctionnement, certaines propriétés telles que **compression**, **quota** et **reservation** par exemple.

5.1 Compression

Il est possible d'indiquer, à la volée, à un système de fichiers ZFS s'il doit compresser les données ou non. Pour l'exemple, créons un système de fichiers nommé **raid6/etc**. En général, ce dossier contient plein de fichiers de configuration de type texte qui se prêtent bien à la compression. Puis, indiquons-lui qu'il doit maintenant compresser les données en mettant la propriété **compression** à **on**. Enfin, copions le contenu de **/etc** et voyons le résultat de la compression :

```
# zfs create raid6/etc
# zfs set compression=on raid6/etc
# zfs get compression raid6/etc
NAME       PROPERTY     VALUE      SOURCE
raid6/etc  compression  on         local
# cp -a /etc/* /media/sauvegarde/etc/
# zfs get compression, compressratio raid6/etc
NAME       PROPERTY       VALUE      SOURCE
raid6/etc  compression    on         local
raid6/etc  compressratio  2.05x      -
# zfs list raid6/etc
NAME       USED  AVAIL  REFER  MOUNTPOINT
raid6/etc  18,2M  26,7G  18,2M  /media/sauvegarde/etc
# du -hs /etc
35M /etc
```

La propriété **compressratio** indique le taux effectif de compression des données, ici un peu plus de 2 fois.

Lorsqu'il existe des données préalablement à la modification de la propriété, ces dernières ne sont pas compressées. Elles restent non compressées sur le disque. Toutefois, et c'est là que c'est intéressant, les nouvelles données ajoutées aux fichiers non compressés, seront, elles, compressées. Ainsi, un fichier peut avoir une partie non compressée et une partie compressée (celle qui aura été écrite après le changement de la propriété).

De même, lorsque l'on modifie à nouveau la propriété vers la valeur **off**, c'est-à-dire sans compression, les données ajoutées le seront sans compression, mais l'état de celles qui étaient compressées précédemment n'est pas modifié.

Cet exemple montre que certaines propriétés affectent les blocs et pas, comme on pourrait s'y attendre, les fichiers eux-mêmes.

5.2 Quotas

Les quotas sur ZFS ne fonctionnent pas à la manière des quotas POSIX. Ils sont fixés par système de fichiers et non par utilisateur. Deux propriétés permettent de les fixer :

- **quota** limite l'espace du système de fichiers, pour lui-même et tous les systèmes de fichiers fils, incluant les instantanés ou snapshots.
- **refquota** fait exactement la même chose, mais la limite n'inclut pas les instantanés ou les descendants.

Il est donc possible de jouer avec ces deux propriétés de manière à limiter la taille maximale occupée par un système de fichiers. Dans notre exemple, on pourrait fixer la taille maximale de **/var** à 5 Go avec la propriété **refquota** et autoriser 5 Go pour les instantanés en fixant à 10 Go la propriété **quota**.

Dès que l'une ou l'autre des propriétés sont définies, la commande **df -h** indique la taille du disque, ainsi que la place restante. Elles sont déterminées par la propriété qui contraint le plus (en général, il s'agit de **refquota**).

Utilisée avec l'option de compression, le quota s'applique, une fois les données compressées. Ainsi, si un système de fichiers est plein avec un quota de 100 Mo, que la propriété de compression est à vrai et que le taux de compression est de 2x, alors ce système de fichiers contient 200 Mo de données non compressées.

Lorsque que le quota est atteint, une erreur survient pour l'indiquer et, dès lors, il n'est plus possible d'ajouter des données ou des fichiers.

5.3 Réservation

La propriété **reservation** permet de réserver de l'espace sur un système de fichiers. Cet espace est directement et immédiatement consommé sur l'espace disponible du système de fichiers parent. Il doit donc être disponible au moment où l'on fixe la valeur de la propriété.

5.4 Héritage des propriétés

Certaines propriétés peuvent être héritées directement de leur(s) parent(s). C'est le cas par exemple pour la compression, le point de montage, les quotas et la réservation. Si l'on change les propriétés d'un système de fichiers, tous ses sous-systèmes hériteront de la nouvelle valeur.

5.5 Instantanés ou snapshots

Comme pour toutes les fonctionnalités que nous avons déjà vues, celle-ci est très simple d'emploi. De plus, la création d'un instantané ne consomme pas de place supplémentaire. Seules les données créées ou modifiées à la suite de cette création utiliseront de l'espace supplémentaire. Un exemple d'utilisation des instantanés :

```
# zfs list raid6/home
NAME         USED  AVAIL  REFER  MOUNTPOINT
raid6/home  57,0K  26,7G  57,0K  /media/sauvegarde/home
# ls -lsa /media/sauvegarde/home
total 25
 3 drwxr-xr-x 2 root root     5 nov 30 19:08 .
 5 drwxr-xr-x 6 root root    10 oct 28 19:43 ..
11 -rw------- 1 root root 10653 nov 30 19:08 .bash_history
 3 -rw------- 1 root root   701 nov 30 19:08 .bash_profile
 3 -rw------- 1 root root  1244 nov 30 19:08 .bashrc
```

Maintenant que nous avons vu ce qu'il y a dans notre répertoire, créons notre premier instantané en utilisant la commande **zfs snapshot nom_du_snapshot** et observons le résultat :

```
# zfs snapshot raid6/home@premier
# zfs list
NAME                USED  AVAIL  REFER  MOUNTPOINT
raid6/home          57,0K  26,7G  57,0K  /media/sauvegarde/home
raid6/home@premier      0      -  57,0K  -
```

Comme on peut le voir, l'instantané **premier** n'occupe aucune place. Copions des données :

```
# cp /etc/passwd /media/sauvegarde/home
# zfs list
NAME                 USED  AVAIL  REFER  MOUNTPOINT
raid6/home           98,9K  26,7G  61,6K  /media/sauvegarde/home
raid6/home@premier   37,2K      -  57,0K  -
```

Il est ainsi possible de créer autant d'instantanés que l'on désire. Nous pouvons par exemple réitérer l'opération : créer un instantané, puis copier un fichier dans le répertoire :

```
# zfs snapshot raid6/home@deuxieme
# cp /etc/X11/xorg.conf /media/sauvegarde/home
```

L'intérêt de tout cela, c'est qu'il est possible de revenir en arrière. Par exemple, je peux revenir à l'état initial en utilisant la commande **zfs rollback raid6/home@premier** ou encore à l'état juste avant la copie de mon dernier fichier avec **zfs rollback raid6/home@deuxieme**. Mieux, il est même possible d'obtenir la différence avec la commande **zfs send -i raid6/home@premier raid6/home@deuxieme > diff**. Dans l'exemple, il s'agit du fichier **/etc/passwd** copié entre les prises des instantanés. Cette différence pourra être relue avec la commande **zfs receive** qui créera alors un instantané correspondant à ce qui lui a été envoyé.

Ces deux dernières commandes peuvent être utilement combinées avec ssh pour, par exemple, générer des sauvegardes différentielles et les envoyer sur un serveur distant.

6 Après un redémarrage

Dans le cas où vous avez redémarré votre machine, votre beau système de fichiers ZFS n'est plus présent. Que faire pour le retrouver ? Comment se souvenir, après quelques jours, mois (ou années) quelle configuration vous aviez utilisée, les disques de spare, raidz1, raidz2, mirror ou pas, le point de montage ?

Avec ZFS, rien de plus simple. Après avoir chargé le module **fuse** et redémarré le démon **zfs-fuse**, rendez-vous directement dans le dossier où vous avez créé vos fichiers (les faux disques) et tapez la commande **zpool import -d .**.

Cette commande recherche, parmi les fichiers du dossier courant (« **.** »), les périphériques virtuels (vdev), puis elle indique le nom et la configuration de chaque pool trouvé parmi ces périphériques. Pour importer réellement un pool, nommez-le à la fin de la commande, par exemple : **zpool import -d . raid6** . Après quelques instants, le système de fichiers est à nouveau monté (sur le bon point de montage) et prêt à être utilisé.

Si le système de fichiers n'est pas monté correctement, vérifiez que vous avez bien le module **fuse** chargé (**modprobe fuse**), que vous avez bien lancé le démon **zfs-fuse** (**zfs-fuse –no-daemon**) et que les répertoires sur lesquels sont montés vos systèmes de fichiers ZFS sont vides.

7 En conclusion

Cet article n'a pas fait le tour de toutes les fonctionnalités et propriétés de ZFS. Aussi, le lecteur curieux pourra regarder de près le système de clones et les permissions liées à l'administration par exemple.

Il est vraiment dommage que ce système de fichiers ne puisse être inclus nativement dans le noyau GNU/Linux par la faute d'une bête incompatibilité de licence...

Bien entendu, tout ce qui est décrit dans cet article est réalisable nativement, c'est-à-dire sans **fuse** et son démon **zfs-fuse**, sous FreeBSD 7.0, Solaris 10, Mac OS X et, me souffle-t-on, Nexenta **[11]**.

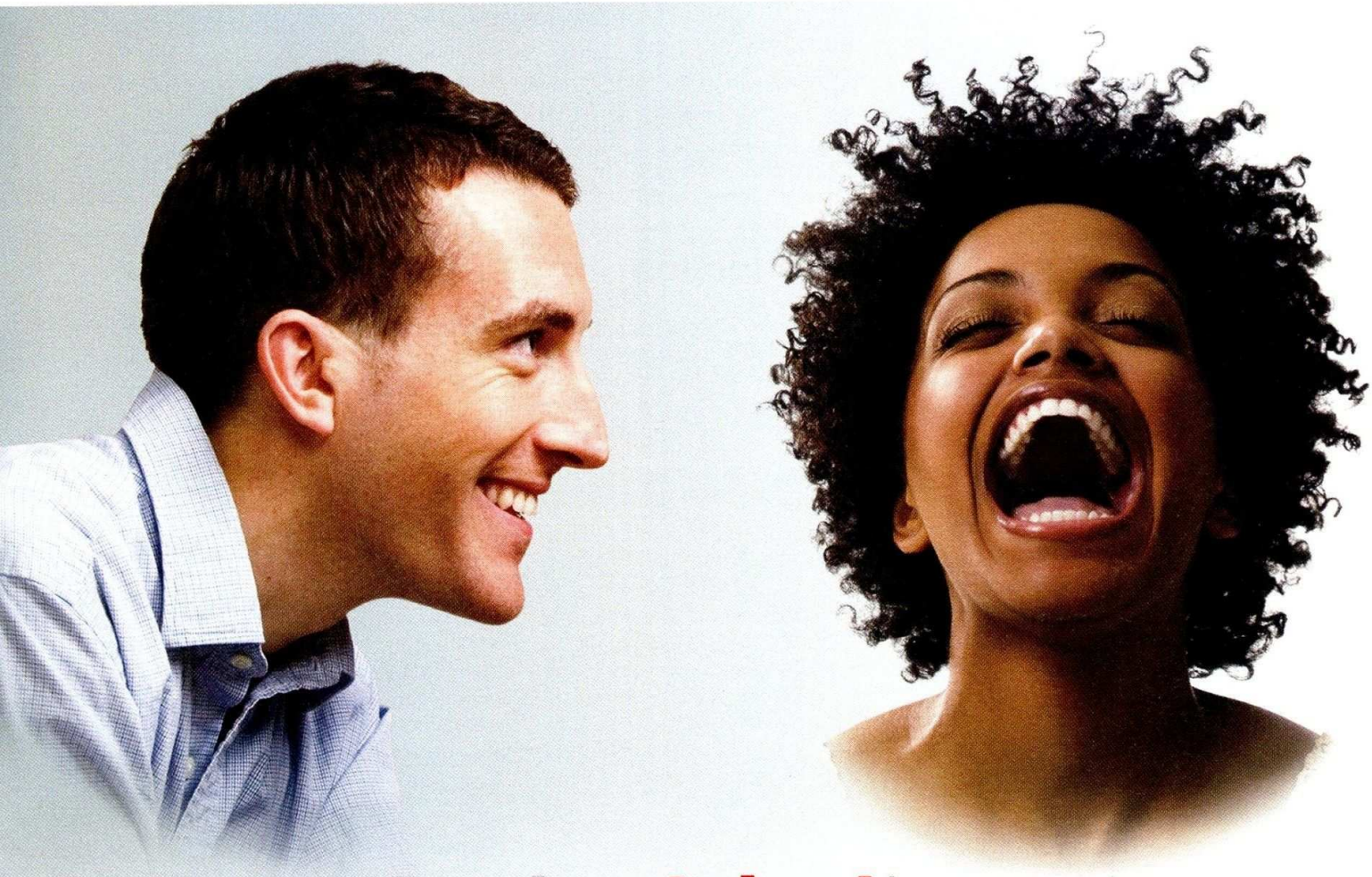
Je tiens à remercier mes mentors qui ont été attentifs aux erreurs et imprécisions et m'ont patiemment relu et corrigé : Bruno Bonfils, Guillaume Lelarge et Sébastien Tricaud.

Auteur : Olivier Delhomme

Références

- [0] http://opensolaris.org/os/community/zfs/source/
- [1] http://opensource.org/
- [2] http://www.opensolaris.org/os/licensing/cddllicense.txt
- [3] http://blogs.sun.com/bonwick/date/20040925#128_bit_storage_are_you
- [4] http://www.freebsd.org/releases/7.0R/announce.html
- [5] http://www.scons.org/
- [6] https://developer.berlios.de/project/showfiles.php?group_id=6836
- [7] http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap12.html#tag_12_02
- [8] http://www.manpagez.com/man/8/zpool/
- [9] http://docs.sun.com/app/docs/doc/817-2271/gazss?l=en&a=view
- [10] http://opensolaris.org/os/community/zfs/docs/
- [11] http://foss-boss.blogspot.com/2008/11/nexenta-can-you-say-solabuntu-part1.html

sagemcommunications



communicative talent*

* L'innovation, notre langage pour exprimer vos talents

Électronique | Logiciel | Réseaux | Télécoms | Radio

Chez Sagem Communications, mettre à profit toute l'intelligence d'une entreprise dédiée aux hautes technologies, c'est aussi faire communiquer les compétences, les idées et les cultures. Ainsi, dans le monde entier, les collaborateurs de Sagem Communications conçoivent, développent, industrialisent et distribuent des produits attractifs et innovants. Au quotidien, leur savoir-faire s'exprime au sein d'équipes dynamiques, pluridisciplinaires et multiculturelles dans une structure qui fait communiquer les talents.

• Ingénieur Linux embarqué h/f

Au sein de l'équipe R&D et en collaboration avec le Chef de projet, vous participez au développement du logiciel embarqué dans des équipements de télécommunications (décodeurs IPMPEG4 HD, Passerelles Triple Play…). Vous définissez les spécifications, les travaux d'architecture et participez à l'implémentation des fonctions demandées selon le cahier des charges du client. **Profil :** Ingénieur Télécom ou informatique, 3 à 5 ans d'expérience en développement logiciel embarqué. Maîtrise du développement driver Linux sur cible et du noyau Linux en environnement embarqué, de l'environnement de développement Linux et des outils de développement Linux GNU. Poste basé à Rueil (92) ou Osny (95). Réf. ILE/SCT

• Chef de projet logiciel communication IP h/f

Au sein de l'équipe R&D, vous devenez leader du pôle d'excellence « IP communications ». Vous êtes force de proposition concernant les fonctions réseau qu'il convient d'intégrer aux produits. Vous concevez et faites réaliser ces fonctions par votre équipe et en gérez la qualité. Vous intervenez également en transverse sur plusieurs projets. **Profil :** Ingénieur ou équivalent, 2 à 5 ans d'expérience dans le domaine du logiciel embarqué comprenant une forte composante « communication IP ». Maîtrise des logiciels sous Linux. Une expérience en management de projets et d'équipes en environnement international serait un plus. Anglais courant. Poste basé à Rueil (92) ou Osny (95). Réf. CDPIP/OSN

• Architecte Logiciel h/f

Au sein des équipes logicielles, vous définissez les architectures dans des environnements logiciels embarqués (généralement sous Linux), la spécification des API et des sous-ensembles logiciels, les choix de solutions et la sélection des souches externes (ex : open source). Vous permettez la mise en œuvre aisée de fonctionnalités innovantes dans nos produits en privilégiant la réutilisation et la portabilité des logiciels. Garant du schéma technique, vous intervenez sur l'ensemble du projet en assurant la mise en application de la conception et de l'architecture. **Profil :** Ingénieur informatique ou équivalent, 3 ans minimum d'expérience en développement de logiciel temps réel. Maîtrise de l'environnement Linux embarqué et expérience en architectures logicielles. Compétences réseaux appréciées. Poste basé à Rueil (92). Réf. AL/OSN

• Ingénieur développement logiciel Linux / Noyau h/f

Au sein de l'équipe R&D, vous réalisez la configuration/optimisation des noyaux, le développement des drivers et le développement des scripts de démarrage. Impliqué sur l'ensemble des phases de nos projets, vous effectuez la rédaction des spécifications internes, le développement, les tests et participez à la qualification et la validation. **Profil :** Ingénieur informatique ou équivalent, 2 ans minimum d'expérience en développement de logiciel temps réel. Vous maîtrisez le développement en C sous Linux embarqué au niveau noyaux/drivers. De bonnes notions en outils de débuggage (GDB) seraient un plus. Poste basé à Rueil (92) ou Osny (95). Réf. IDLL/OSN

Pour rejoindre nos équipes dynamiques à forte culture technologique, merci de nous faire parvenir par mail votre candidature, en indiquant la référence choisie, à : rh_osn@sagem.com. Consultez l'ensemble de nos offres d'emplois en France et à l'international, stages, VIE et opportunités d'expatriation à la rubrique carrières de notre site **www.communicative-talent.com**

Mise en œuvre de phpCAS

Cet article présente la mise en place d'un système d'authentification SSO (Single Sign-On) de type CAS (Central Authentification Service) sur un ou plusieurs serveurs WEB utilisant le langage PHP.

Auteur

- Christophe Borelly

1 Introduction

L'article précédent (*GNU/Linux magazine* n° 113) présentait la boite à outils CAS-Toolbox **[6]** qui permet de déployer et de personnaliser un service d'authentification CAS **[5]**. Ici, nous allons étudier la bibliothèque phpCAS **[1]** qui permet d'utiliser le service CAS en PHP.

Note

Je me dois ici de réparer le petit oubli de mon premier article en précisant le nom des personnes qui ont œuvré pour que ces projets existent :

- Pascal AUBRY (Université de Rennes 1) - auteur de phpCAS.
- Julien MARCHAL (Université de Nancy 2) - auteur de CAS-Toolbox.

Il ne faut pas oublier également toute la communauté du consortium ESUP-Portail (http://www.esup-portail.org/) qui est à l'origine du développement et de la diffusion des projets utilisant CAS dans le monde universitaire français depuis 2003.

Je voudrais aussi préciser que le projet CAS-Toolbox est bien géré par le consortium ESUP-Portail et qu'il est seulement hébergé sur la plate-forme SourceSup du CRU.

Évidemment, il faut avoir à disposition un serveur WEB avec le support de PHP pour pouvoir programmer des services PHP sécurisés. La majorité des distributions LINUX proposent des paquets binaires déjà prêts pour le serveur WEB Apache httpd et l'interpréteur PHP. Il faut néanmoins vérifier que ces paquets sont conformes aux prérequis **[2]** de la bibliothèque phpCAS :

- Apache (version minimum 2.0.44) avec le module SSL ;
- Bibliothèque cURL (version minimum 7.5) avec le support SSL ;
- PHP (version minimum 4.3.1) avec DOM, ZLIB, cURL, OPENSSL et PEAR::DB.

Cela peut se faire rapidement avec les commandes suivantes. Notez la petite subtilité pour httpd avec l'utilisation d'une redirection du flot d'erreurs sur la sortie standard (**2>&1**) pour pouvoir y rechercher des chaines avec **egrep** :

```
/usr/local/apache2/bin/httpd -M 2>&1 | egrep "ssl|php"
ssl_module (static)
php5_module (shared)
curl -V
curl 7.16.2 (i686-pc-linux-gnu) libcurl/7.16.2
OpenSSL/0.9.8i zlib/1.2.3 libidn/1.5
Protocols: tftp ftp telnet dict ldap http file https
ftps
Features: IDN IPv6 Largefile NTLM SSL libz
php -i | egrep "cURL|DOM|SSL|ZLib"
cURL support => enabled
cURL Information => libcurl/7.16.2 OpenSSL/0.9.8i
zlib/1.2.3 libidn/1.5
DOM/XML => enabled
DOM/XML API Version => 20031129
OpenSSL support => enabled
OpenSSL Version => OpenSSL 0.9.8i 15 Sep 2008
ZLib Support => enabled
```

2 Mise en place de phpCAS

Le client phpCAS peut s'installer de façon manuelle en dé-compactant l'archive **CAS-1.0.1.tgz** **[1]** dans le répertoire contenant les bibliothèques PHP (**/usr/local/lib/php** sur ma machine). Ensuite, il faut modifier ou ajouter dans le fichier **php.ini** la prise en compte de l'**include_path** vers la ligne 469 : **include_path = ".:/usr/local/lib/php"**.

Après une modification du fichier **php.ini**, un redémarrage du service httpd (**httpd -k restart**) s'impose. À partir de là, tout est prêt pour tester la bibliothèque phpCAS.

L'exemple qui suit suppose que le serveur CAS s'appelle **cas.iutbeziers.fr** et qu'il utilise le port SSL 8443. À la ligne 7, la méthode qui indique au script de ne pas vérifier le certificat X.509 envoyé par le serveur CAS. Elle est utilisé ici simplement pour faciliter le test de la bibliothèque CAS. Nous verrons plus loin comment préciser le certificat d'autorité à utiliser pour vérifier correctement le certificat du serveur CAS.

La méthode la plus importante est celle de la ligne 8 qui oblige les utilisateurs à s'authentifier. Le reste du code PHP n'est pas exécuté tant que ce n'est pas le cas.

On peut noter au passage à la ligne 10, la méthode **phpCAS::getUser()** qui, comme son nom l'indique, permet de récupérer le nom de l'utilisateur connecté.

```
01 <?php
02 require_once('CAS-1.0.1/CAS.php');
03 define('LF',"\n");
04 // CAS se trouve à la racine du site, donc le dernier paramètre = ''
05 phpCAS::client(CAS_VERSION_2_0,'cas.iutbeziers.fr',8443,'');
06 // Ne pas vérifier le certificat du serveur CAS
07 phpCAS::setNoCasServerValidation();
08 phpCAS::forceAuthentication();
09 echo '<h1>Authentification réussie sur '.$_SERVER['HTTP_HOST'].'
(client CAS) !</h1>'.LF;
10 echo '<h2>Utilisateur : '.phpCAS::getUser().'</h2>'.LF;
11 ?>
```

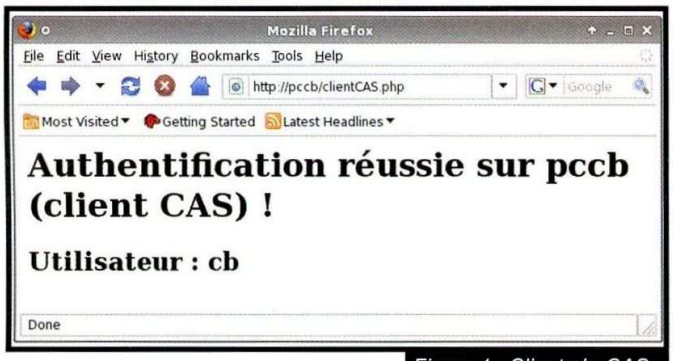


Figure 1 : Client phpCAS

Si on écrit un script équivalent sur un autre serveur Apache (ou avec un **VirtualHost** **[11]**), on peut vérifier que l'authentification est « transparente » lorsque l'on tape l'URL du second script.

Figure 2 : Vérification de l'authentification SSO

2.1 Utilisation avancée

Une fonction de débogage existe (**phpCAS::setDebug()**). Elle peut servir à analyser la cause d'une erreur de fonctionnement ou à étudier le processus interne de phpCAS. Cette fonction crée le fichier **/tmp/phpCAS.log** et ajoute toute l'information nécessaire pour une analyse future. Voici un petit exemple de ce que cela peut donner :

```
5F85 .START ****************** [CAS.php:414]
5F85 .=> phpCAS::setNoCasServerValidation() [clientCAS.php:7]
5F85 .<= ''
5F85 .=> phpCAS::forceAuthentication() [clientCAS.php:8]
5F85 .|    => CASClient::forceAuthentication() [CAS.php:911]
5F85 .|    |    => CASClient::isAuthenticated() [client.php:686]
5F85 .|    |    |    => CASClient::wasPreviouslyAuthenticated()
                       [client.php:791]
5F85 .|    |    |    |    no user found [client.php:895]
5F85 .|    |    |    <= false
5F85 .|    |    |    no ticket found [client.php:825]
5F85 .|    |    <= false
5F85 .|    |    => CASClient::redirectToCas(false) [client.php:695]
5F85 .|    |    |    => CASClient::getServerLoginURL(false, false)
                       [client.php:912]
5F85 .|    |    |    |    => CASClient::getURL() [client.php:331]
5F85 .|    |    |    |    <= 'http://pccb/clientCAS.php'
5F85 .|    |    |    <= 'https://pccb:8443/login?service=http%3A%2F%2Fp
                        ccb%2FclientCAS.php'
5F85 .|    |    |    Redirect to : https://pccb:8443/login?service=http
                                    %3A%2F%2Fpccb%2FclientCAS.php
5F85 .|    |    |    exit()
5F85 .|    |    |    -
5F85 .|    |    -
5F85 .|    -
```

La bibliothèque comporte également plusieurs fonctions **[3]** qui permettent d'obtenir par exemple des URL d'accès au service CAS ou bien l'URL du service actuel (**getServiceURL()**) :

```
…
10 echo '<h2>Utilisateur : '.phpCAS::getUser().'</h2>'.LF;
11 echo '<h2>getServerLoginURL : '.phpCAS::getServerLoginURL().'</h2>'.LF;
12 echo '<h2>getServerLogoutURL : '.phpCAS::getServerLogoutURL().'</h2>'.LF;
13 echo '<h2>getServiceURL : '.phpCAS::getServiceURL().'</h2>'.LF;
14 ?>
```

À l'exécution, on obtient cela :

```
…
Utilisateur : cb
getServerLoginURL : https://cas.iutbeziers.fr:8443/login?service=http%3
A%2F%2Flocalhost%2FclientCAS.php
getServerLogoutURL : https://cas.iutbeziers.fr:8443/logout
getServiceURL : http://localhost/clientCAS.php
```

La fermeture de la session CAS se fait avec les commandes **phpCAS::logout()**, **phpCAS::logoutWithUrl()** et **phpCAS::logoutWithRedirectService()**. Les différences entre elles se situent au niveau de la page finale après la déconnexion.

La première fonction ne fait rien de particulier si ce n'est de rediriger l'utilisateur sur la *servlet* de *logout*.

La seconde précise l'URL de destination une fois déconnecté du service CAS. L'utilisateur se retrouve encore sur la page de logout du serveur CAS (pour qu'il puisse vérifier qu'il a bien été déconnecté), mais, cette fois-ci, il peut voir un message supplémentaire lui donnant le lien vers l'URL précisée plus haut.

Enfin, la dernière fonction permet automatiquement après la déconnexion de demander une nouvelle connexion vers un autre service.

On peut donc ajouter, dans le script **clientCAS.php**, le code suivant pour la déconnexion (notez le lien de déconnexion à la ligne 20) :

```
…
08 if (isset($_REQUEST['logout']))
09 {
10    //phpCAS::logout();
11    phpCAS::logoutWithUrl('http://localhost/');
12    //phpCAS::logoutWithRedirectService('http://pccb/clientCAS.php');
13 }
…
20 echo '<a href="'.$_SERVER['PHP_SELF'].'?logout">Déconnexion</a>'.LF;
```

3 Configuration d'un proxy phpCAS

La fonction proxy CAS permet de créer un ticket utilisable pour accéder à un autre service CAS. Cette fonctionnalité est souvent utilisée dans les ENT (Environnements Numériques de Travail) qui rassemblent dans une même interface plusieurs services différents.

Par exemple, si on veut qu'un script **a.php** sur le serveur **A** puisse afficher le contenu du service **b.php** (qui peut être sur le serveur A ou non), il faut dans ce cas-là que le script **a.php** serve de proxy CAS pour le service fourni par le script **b.php**.

Par construction, le système impose que le proxy soit sécurisé par SSL/TLS, car le serveur CAS doit échanger des données avec lui **[4]**. Il faut donc, pour cela, ajouter le support de SSL/TLS au serveur Apache et, de plus, indiquer, à l'environnement d'exécution JAVA, le certificat d'autorité utilisé par ce serveur sous peine qu'il ne puisse pas le contacter (JAVA vérifie automatiquement la validité des certificats SSL et compare également le nom du certificat au nom du serveur utilisé).

3.1 Installation d'Apache en mode SSL

On suppose ici que l'on a un certificat de serveur SSL pour le nom **pccb** (fichiers **server-httpd.crt** et **server-httpd.key**) émis par l'autorité de certification utilisée dans l'article précédent (**ca.crt**).

L'activation du port HTTPS 443 se fait en éditant le fichier **/usr/local/apache2/conf/httpd.conf**. Vers la fin de ce fichier, il faut enlever les commentaires devant **Include conf/extra/httpd-ssl.conf** et, ensuite, modifier ce fichier **httpd-ssl.conf** pour utiliser les fichiers de certificat et la CA comme indiqué ci-dessous.

Pour ne pas avoir de message d'erreur dans **/usr/local/apache2/logs/error_log** du style « **[warn] RSA server certificate CommonName (CN) 'pccb' does NOT match server name!?** », il faut également modifier le champ **ServerName**, pour qu'il corresponde au champ CN du certificat :

```
…
ServerName pccb
…
SSLCertificateFile conf/server-httpd.crt
SSLCertificateKeyFile conf/server-httpd.key
SSLCACertificateFile conf/ca.crt
…
```

Après avoir copié les fichiers de certificat dans **/usr/local/apache2/conf**, on peut tester le fichier de configuration avec l'option **-t** de **httpd**, puis relancer le démon avec l'option **-k restart**. Une connexion sur **https://pccb/** permet de vérifier le fonctionnement de HTTPS.

```
/usr/local/apache2/bin/httpd -t
Syntax OK
/usr/local/apache2/bin/httpd -k restart
tail /usr/local/apache2/logs/error_log
…
[Wed Nov 4 18:24:18 2008] [notice] Apache/2.2.10 (Unix) mod_ssl/2.2.10
OpenSSL/0.9.8i PHP/5.2.6 configured -- resuming normal operations
…
```

3.2 Installation du certificat de CA pour JAVA sur le serveur CAS

Cette étape a déjà été faite lors du dernier article sur CAS-Toolbox, quand il a fallu tester l'authentification LDAPS sur le serveur CAS. Je la redonne ici pour conserver l'unité de l'article.

La liste des certificats de confiance de JAVA se trouve dans le fichier **$JAVA_HOME/jre/lib/security/cacerts**. L'utilitaire JAVA **keytool** permet de gérer son contenu. On peut exécuter les commandes suivantes :

```
JAVA_STORE=$JAVA_HOME/jre/lib/security/cacerts
keytool -import -alias ca -keystore $JAVA_STORE -storepass changeit \
-trustcacerts -file /usr/local/apache2/conf/ca.crt
…
SubjectAlternativeName [
  CN=CA, O=IUT, L=BEZIERS, ST=HERAULT, C=FR
]

Faire confiance à ce certificat ? [non]:  oui
Certificat ajouté au Keystore
```

La vérification de la liste des certificats de confiance se fait avec l'option **-list** :

```
keytool -list -alias ca -keystore $JAVA_STORE -storepass changeit
ca, 4 nov. 2008, trustedCertEntry,
Empreinte du certificat (MD5) : AE:3E:3F:00:8F:DD:A4:62:FB:0D:6E:68:70
:32:D1:BC
```

3.3 Test du proxy

Le code source est pratiquement le même que pour le client, si ce n'est que l'on utilise la méthode **phpCAS::proxy()**. On peut noter aussi la vérification du certificat envoyé par le serveur CAS avec la méthode **phpCAS::setCasServerCACert()**. Le service que l'on veut utiliser est interrogé par la méthode **phpCAS::serviceWeb()** et le résultat est renvoyé dans la variable **$output** :

```php
<?php
require_once('CAS-1.0.1/CAS.php');
define('LF',"\n");
// Active les logs de phpCAS voir /tmp/phpCAS.log
//phpCAS::setDebug();
phpCAS::proxy(CAS_VERSION_2_0,'cas.iutbeziers.fr',8443,'');
//phpCAS::setNoCasServerValidation();
phpCAS::setCasServerCACert('/usr/local/apache2/conf/ca.crt');
// Sous WINDOWS, il faut préciser le répertoire de sauvegarde du ticket PGT
//phpCAS::setPGTStorageFile('','C:/Tmp');
phpCAS::forceAuthentication();
echo '<h1 >Authentification réussie sur '.$_SERVER['HTTP_HOST'].'
(proxy CAS) !</h1>'.LF;
echo '<h2 >Utilisateur : '.phpCAS::getUser().'</h2>'.LF;

$service='http://localhost/clientCAS.php';
// Appel du service
if (phpCAS::serviceWeb($service,$err_code,$output))
{
   echo '<font style="color:green;">';
}
else
{
  echo '<font style="color:red;">';
}
echo $output;
echo '</font>';
?>
```

Le résultat est le suivant (notez en vert le source du service demandé par le proxy CAS) :

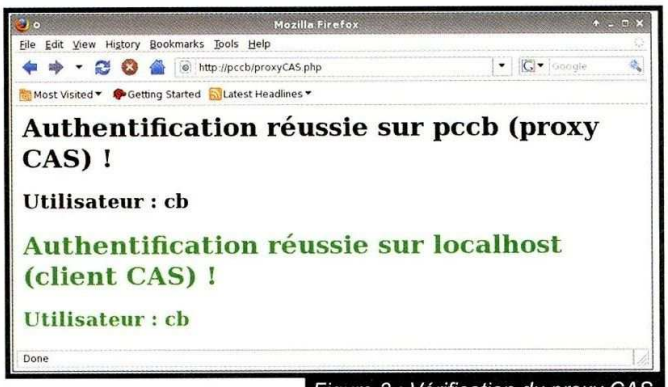


Figure 3 : Vérification du proxy CAS

4 Conclusion

Voilà les bases de l'utilisation de la bibliothèque phpCAS expliquées. Dans de prochains articles, je vous monterai comment utiliser cette bibliothèque pour l'appliquer à un client de messagerie (SquirrelMail), puis à un portail captif (ChilliSpot).

Auteur : Christophe Borelly

Professeur de l'ENSAM

Département Réseaux et Télécommunications

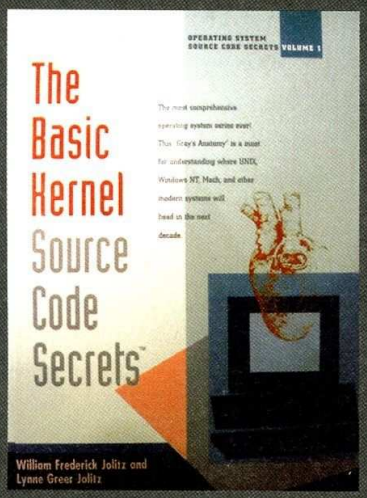
IUT de Béziers

Liens

- [1] Client phpCAS : http://www.ja-sig.org/wiki/display/CASC/phpCAS
- [2] Pré-requis de phpCAS : http://www.ja-sig.org/wiki/display/CASC/phpCAS+requirements
- [3] Documentation phpCAS : http://www.ja-sig.org/downloads/cas-clients/php/1.0.1/docs/api/classphpCAS.html
- [4] Présentation de CAS – Vincent Mathieu, Pascal Aubry, Julien Marchal, JRES2003, Lille, novembre 2003 : http://perso.univ-rennes1.fr/pascal.aubry/presentations/cas-jres2003/
- [5] Site du projet JA-SIG : http://www.ja-sig.org/products/cas/
- [6] Site du projet CAS-Toolbox : http://sourcesup.cru.fr/projects/cas-toolbox/
- [7] OpenSSL : http://www.openssl.org/
- [8] OpenLDAP : http://www.openldap.org/
- [9] cURL: http://curl.haxx.se/
- [10] Apache httpd : http://httpd.apache.org/
- [11] VirtualHost : http://httpd.apache.org/docs/2.2/fr/vhosts/examples.html
- [12] PHP : http://www.php.net/

LIVRE(S)

Critique : Source Code Secrets : The Basic Kernel



Informations

- Auteurs : Lynne G. Jolitz & William F. Jolitz
- Broché 530 pages
- Éditeur : Coriolis Group Books (juillet 1996)
- Langue : Anglais
- ISBN-10: 1573980269
- ISBN-13:978-1573980265

Auteur

- Par Emile iMil Heitor, pour GCU canal historique

Sur Wikipédia, dans le résumé du roman *L'Histoire sans fin*, on peut lire ceci :

« Bastien, un jeune garçon, emprunte en douce un livre intitulé *L'Histoire sans fin* dans une librairie après que le libraire lui a dit que ce livre était dangereux. Au fur et à mesure qu'il avance dans la lecture du livre, il se retrouve lui-même faisant partie de la quête dont le but est de sauver le monde et les habitants du Pays Fantastique. »

Si je devais faire un résumé du livre, que je qualifierais d'ailleurs de roman, *The Basic Kernel: Source Code Secrets*, je n'écrirais pas mieux.

William Frederick Jolitz et Lynne Greer Jolitz font partie de cette catégorie de hackers finalement peu connus qui ont pourtant proprement révolutionné le cours de l'histoire de l'informatique. Ce couple d'illuminés n'est ni plus ni moins que l'auteur du portage de 4.3BSD-Reno vers l'architecture i386, préparant ainsi le terrain pour deux projets qui connaîtront le succès que 386BSD aura peu ou pas connu : NetBSD et FreeBSD.

The Basic Kernel: Source Code Secrets raconte l'histoire de ce portage. Fichier par fichier. Si ! En effet, Lynne et William Jolitz ont eu la bonne idée, pendant leur périple, de noter les nombreuses étapes de ce portage hors du commun. Ainsi, après un premier chapitre quasi biblique, nous sommes plongés dès le second chapitre (*Assembly entry and primitives*) dans les méandres du *bootstrap* à grands renforts de schémas explicatifs et codes parfaitement commentés. Inutile de préciser que quelques notions d'architecture et de bons restes d'assembleur seront nécessaires à la bonne compréhension de cette introduction. Tout au long du livre, chaque étape est décortiquée sous forme de question/réponse :

What is XXX ?

How is XXX implemented ?

Un vrai régal.

À partir du troisième chapitre, *CPU Specific primitives*, les amoureux du C seront probablement captivés par la mécanique de naissance du noyau, partie évidemment encore également intimement liée à l'architecture. Suit une immersion dans les services fournis par le noyau (*Internal kernel services*), où l'on retrouvera l'ancêtre d'**`autoconf(4)`** (non, l'autre), **`config.c`**, responsable de l'interfaçage du matériel. Dans cette section, on apprendra également comment les inséparables **`malloc(3)`** et **`free(3)`** furent implémentés. Le chapitre cinq explique en détail le cycle de vie d'un processus par le biais de l'étude du code de **`fork(2)`**, **`exit(3)`** et appels connexes, puis nous apprendrons comment est implémenté le mécanisme de signaux POSIX dans **`sig.c`**. Dans le chapitre *Credentials and privileges*, la compréhension de **`cred.c`** et **`priv.c`** nous éclairera sur les classiques **`setuid()`**, **`setgid()`** et l'ensemble des fonctions liées aux privilèges des utilisateurs dans un système UNIX. La section *Process Multiplexing* n'est pas des plus digestes, puisqu'on y apprend les algorithmes de *scheduling* ou comment le noyau répartit la charge entre les processus. Enfin, *POSIX operating system functionnality* entre dans le détail des fonctions que chaque programmeur C doit manipuler plusieurs centaines de fois par semaine, **`execve()`**, **`dup()`**, **`open()`**, **`read()`**, **`close()`** et j'en passe. Ces bijoux trouvent leurs ressources dans **`execve.c`** et **`descript.c`**.

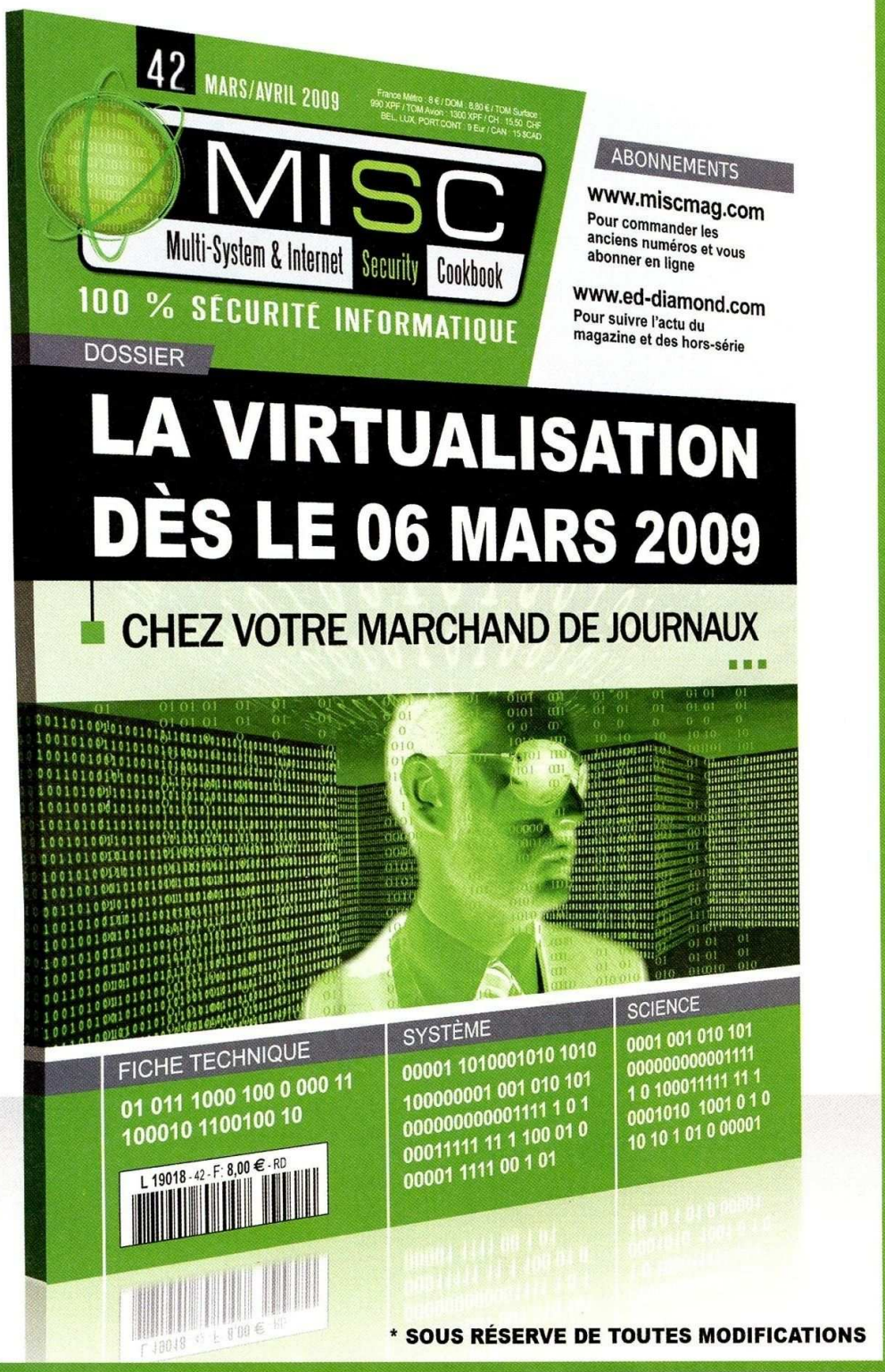
Vous l'aurez compris, ce grimoire n'est pas à mettre entre toutes les mains, et de solides connaissances en langage C sont nécessaires à sa bonne appréhension. Cependant, au travers de cette saine lecture, c'est tout le fonctionnement, au plus bas niveau, de votre système d'exploitation UNIX ou UNIX-like que vous serez à même de décrypter, car, si cet ouvrage décrit des évènements qui ont eu lieu voilà une quinzaine d'années, vous serez surpris de découvrir que les concepts ancestraux sont tout à fait d'actualité.

The Basic Kernel: Source Code Secrets ne semble plus être distribué. Aussi, il vous faudra avoir recours aux divers sites de vente par correspondance type Amazon.fr pour vous en procurer un exemplaire.

Oh, et pour finir : navré pour vos nuits.

LA VIRTUALISATION, UNE SOLUTION VRAIMENT SÛRE ?

MISC 42



* SOUS RÉSERVE DE TOUTES MODIFICATIONS

AU SOMMAIRE...

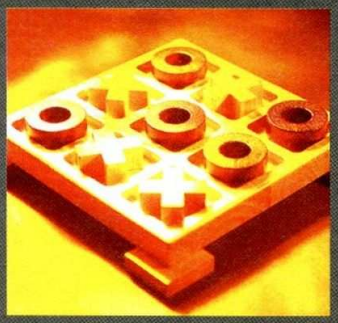
- Vers une version française de la guerre de l'information ?
- WPA en ligne de mire : La sécurité du WiFi à nouveau sur la sellette
- Programmation : l'obfuscation contournée
- Plus loin dans la sécurité des clés USB
- Émulation d'architectures réseau : présentation de Dynamips/Dynagen/GNS3
- Sûreté de fonctionnement et sécurité des algorithmes cryptographiques

Dossier :

- Introduction à la virtualisation
- Voyage au centre de Xen : Découvrez les mécanismes d'un des systèmes de virtualisation les plus utilisés
- Bonnes pratiques pour la virtualisation avec VMware ESX
- Inside Microsoft Hyper-V
- Détection opérationnelle des rootkits Hardware-based Virtual Machine

DISPONIBLE DÈS LE 06 MARS 2009 CHEZ VOTRE MARCHAND DE JOURNAUX

GNU/Linux sur PlayStation



La Playstation Portable (PSP) est un ordinateur, généralement utilisé comme console de jeu, à base de processeurs MIPS[1], fournissant quelques interfaces avec l'utilisateur (boutons, port série, USB et wifi), un clavier de bonne qualité et plus de 32 MB de RAM sans MMU[2]. Il s'agit donc d'un environnement idéal pour faire tourner uClinux. Nous basant sur divers projets actifs sur le web, nous présentons les étapes pour installer un système uClinux fonctionnel sur cette plateforme, et l'ajout d'une interface avec clavier PS2 pour faciliter les développements. Il s'agit là de bases qui doivent encourager le lecteur à contribuer au portage d'uClinux sur PSP, puisque le support de nombreux périphériques est encore absent du noyau.

Auteurs

- S. Guinot
- J.-M. Friedt

Notes

[1] Une des architectures de processeur les plus courantes, issues des recherches à l'université de Stanford, en opposition notamment avec l'architecture SPARC développée à Berkeley.

[2] L'absence de gestionnaire de mémoire a des implications quant aux fonctionnalités supportées par le matériel, notamment concernant l'efficacité de changements de contexte lorsque le *scheduler* passe d'un processus à l'autre, allocation de mémoire ou création de processus. Ces différences expliquent la création d'une branche dédiée du noyau Linux pour ces architectures : **uClinux à www.uclinux.org**.

La Playstation Portable (http://en.wikipedia.org/wiki/PlayStation_Portable, PSP) fournit une plateforme contenant un processeur généraliste basé sur une architecture MIPS R4000 (architecture `-mips3` de `gcc`), 32 MB de RAM et la capacité à exécuter un jeu depuis son support de stockage de masse non volatile (*memory stick*). Il s'agit donc d'un environnement presque idéal pour installer une version embarquée de GNU/Linux sur système sans gestionnaire de mémoire : uClinux. Des progrès ont récemment été fait en ce sens, que nous proposons de présenter ici. Nous allons développer la mise en œuvre d'une *toolchain* de *cross-compilation* pour générer le code à destination de la PSP, et le fonctionnement du *bootloader* qui se présente comme un jeu exécuté depuis la carte de stockage *Memory Stick* : ces outils sont disponibles sous forme d'archive afin de faciliter leur obtention. Afin de rendre la programmation sur PSP fonctionnelle, nous nous proposerons d'ajouter une interface PS2-RS232 afin de connecter un clavier au port série de la PSP et ainsi disposer d'une interface de communication pratique.

1 Les toolchains

Nous allons présenter deux ensembles d'outils de compilation : dans un premier temps, les outils pour compiler le bootloader qui apparaît comme un jeu pour la PSP, et, d'autre part, les outils pour compiler le noyau uClinux et les outils associés. La première partie n'est réellement intéressante que pour le développeur désireux de modifier le bootloader et y ajouter ses propres fonctions.

Le second ensemble d'outils est utile pour compiler le noyau uClinux et les outils associés pour l'utilisateur (principalement Busybox). Contrairement au format d'exécutables ELF habituellement utilisé par GNU/Linux sur plateforme disposant d'un gestionnaire matériel de mémoire, uClinux utilise le format d'exécutables *binary flat* (BFLT). Il est basé sur le format `a.out`, et fournit un sous-ensemble des fonctionnalités proposées par le format ELF : plus petit et plus rapide à charger, les exécutables dans ce format sont mieux adaptés aux systèmes embarqués aux ressources réduites.

La chaîne de cross-compilation MIPS – fonctionnelle puisqu'un certain nombre

de routeurs exploitent des processeurs basés sur cette architecture **[1]** – doit être modifiée afin de générer des binaires FLAT au lieu des habituels ELF. Pour cela, un éditeur de liens spécial doit être utilisé : **elf2flt**, indisponible à l'origine pour architecture MIPS **[2]**.

Le « paquet » **elf2flt** est composé des éléments suivants :

- le script shell **ld-elf2flt** ;
- le programme **elf2flt** capable de convertir des binaires ELF au format BFLT ;
- le programme **flthdr** qui permet d'éditer les en-têtes FLAT.

Le script **ld-elf2flt** va venir se substituer au binaire **ld** original. **ld** est lui renommé en **ld.real**. Si le script **ld** est invoqué avec l'option **-elf2flt**, alors **ld.real**, puis **ld-elf2flt** sont successivement utilisés pour générer un binaire au format ELF, puis au format BFLT. Dans le cas contraire, **ld.real** est utilisé seul. En conséquence, le comportement de la toolchain n'est modifié que lorsque l'option **-elf2flt** est passée à l'éditeur de liens. Exemple de compilation :

```
# mipsel-psp-gcc -Wl,-elf2flt -o test test.c
# ls
test.gdb test
```

À noter que le programme **elf2flt** produit deux fichiers. Le binaire avec le suffixe **.gdb** contient des informations utiles au débogage et peut être utilisé par exemple avec **gdbserver**. Le format FLAT n'embarque que quelques sections indispensables à l'exécution d'un binaire : **.text**, **.data** et **.bss**. Il ne contient donc pas les sections de débogage que l'on trouve habituellement dans un binaire ELF.

En résumé, la principale difficulté pour construire une toolchain MIPS-PSP est d'inclure les programmes **elf2flt** à la suite Binutils.

Pour mieux comprendre le format BFLT, une bonne source est le *loader* du noyau GNU/Linux. Son code se trouve dans le fichier **fs/binfmt_flat.c**. Une autre source incomplète, mais utile, est le document http://www.beyondlogic.org/uClinux/bflt.htm.

1.1 La compilation de jeux pour PSP

Nous avons déjà décrit dans ces pages [3] la modification logicielle à effectuer sur sa PSP pour pouvoir exécuter des jeux depuis la carte mémoire et compiler ses propres jeux au moyen de la PSP toolchain[3] et du PSP SDK[4]. Avec l'avènement de nouveaux *firmwares* sur les PSP plus récentes que janvier 2007 (date de rédaction du précédent article) et de la nouvelle PSP Slim, le lecteur se reportera au web pour les dernières nouvelles à ce sujet. La compilation du bootloader et surtout la lecture du code associé nous semblent cependant instructive pour comprendre comment exécuter un noyau Linux sur toute plateforme sur laquelle un système tourne déjà. Le bootloader décrit ici (Fig. 1) a été écrit par J. Mo **[4]**, et ne diffère de celui de C. Mulhearn **[5]** que par sa capacité à charger une image compressée par gzip : il faudra donc penser à compiler la zlib pour PSP telle que décrite à http://www.psp-programming.com/tutorials/c/lesson04.htm.

Notes

[3] http://ps2dev.org/psp/Tools/Toolchain/psptoolchain-20070626.tar.bz2

[4] http://ps2dev.org/psp/Projects/PSPSDK

À noter que, au cours de l'exécution du bootloader, les routines du système d'exploitation Sony natif (que nous visons à remplacer par GNU/Linux) sont encore disponibles, et, avec elles, l'usage de fonctions telles que l'initialisation des ports série ou du cache du processeur qui ne sont pas nécessairement documentées (ou désassemblées) pour être accessibles sous Linux. C'est donc le dernier moment, avant de lancer Linux, pour initialiser les périphériques dont la gestion n'est possible que par le système d'exploitation Sony (par exemple l'affichage en mode texte à l'écran est encore accessible par **pspDebugScreenPrintf()**).

Le code qui suit s'occupe de charger le fichier contenant l'image, de le décompresser, de le copier en RAM dont l'adresse commence en 0x88000000 **[6]**, d'activer les divers périphériques (port série associé au casque audio et cache du processeur), pour finalement exécuter les instructions à partir du début de la RAM.

```
#define KERNEL_ENTRY        0x88000000
#define KERNEL_PARAM_OFFSET 0x00000008
#define KERNEL_MAX_SIZE     (size_t)( 4 * 1024 * 1024 )   /* 4M */

#define printf              pspDebugScreenPrintf

BOOL loadKernel(void ** buf_, int * size_)
{
  gzFile zf;
  void * buf;
  int size;

  zf = gzopen( s_paramKernel, "r" );
  buf = (void *)malloc( KERNEL_MAX_SIZE );
  size = gzread( zf, buf, KERNEL_MAX_SIZE );
  gzclose( zf );
  *buf_ = buf;
  *size_ = size;
}
/*-------------------------------------------------------------------*/
void transferControl(void * buf_, int size_)
{
    KernelEntryFunc kernelEntry = (KernelEntryFunc)( KERNEL_ENTRY );

    /* prepare kernel image */
    memCopy( (void *)( KERNEL_ENTRY ), buf_, size_ );

    uart3_setbaud( s_paramBaud );
    uart3_puts( "Booting Linux kernel…\n" );

    kernelEntry( 0, 0, kernelParam );
}
```

Le bootloader (Fig. 1, page suivante) – disponible notamment dans l'archive associée à cet article sur la page web des auteurs à http://jmfriedt.free.fr/ – se compile donc comme un jeu pour PSP par **make kxploit** pour fournir les deux répertoires classiques **pspboot** et **pspboot%**.

```
$ make kxploit
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -D_PSP_
FW_VERSION=150   -c -o main.o main.c
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -I. -I/usr/
local/pspdev/psp/sdk/include -O2 -G0 -Wall   -c -o kmodlib.o kmodlib.S
psp-gcc -I. -I/usr/local/pspdev/psp/sdk/include -O2 -G0 -Wall -D_PSP_
FW_VERSION=150  -L. -L/usr/local/pspdev/psp/sdk/lib   main.o kmodlib.o
-lz -lpspdebug -lpspdisplay -lpspge -lpspctrl -lpspsdk -lc -lpspnet
-lpspnet_inet -lpspnet_apctl -lpspnet_resolver -lpsputility -lpspuser
-lpspkernel -o pspboot.elf
"
```

Figure 1: Le bootloader permettant d'exécuter un système GNU/Linux sur PSP se présente comme un jeu, lancé depuis la carte mémoire Memory Stick. Un espace total de l'ordre de 1,5 MB est nécessaire sur la carte.

Recompiler son propre bootloader est l'occasion d'en étudier le fonctionnement : la copie en mémoire volatile (RAM) du système d'exploitation Sony est écrasée pour être remplacée par l'image en *raw binary* de uClinux. Ce code est directement exécutable sur l'architecture cible, sous réserve d'avoir été compilé pour un emplacement approprié en RAM : nous trouverons dans **target/device/mips/psp/linux26.config** la définition de cette adresse sous forme de :

```
CONFIG_PSP_ADDRESS_BASE=0x80000000
jmfriedt@(none):/mnt/sde1/psp/game$ ls -l pspboot
total 1056
-rwxr-xr-x 1 root root 163636 2008-01-04 16:09 eboot.pbp
-rwxr-xr-x 1 root root    522 2008-01-26 16:26 pspboot.conf
-rwxr-xr-x 1 root root 860798 2008-05-08 15:12 vmlinux.bin
jmfriedt@(none):/mnt/sde1/psp/game$ ls -l pspboot%
total 32
-rwxr-xr-x 1 root root 8292 2008-01-04 16:09 eboot.pbp
```

Nous placerons, dans le même répertoire de la carte MemoryStick, l'image uClinux complète contenant le noyau et le système de fichiers, ainsi que le bootloader associé.

1.2 Les outils pour compiler uClinux pour MIPS

Nous nous sommes efforcés de fournir un environnement cohérent (Fig. 2) pour compiler le noyau uClinux et les applications associées, notamment Busybox qui fournira tous les outils GNU habituels. Nous avons dans ce but utilisé et modifié le framework de cross-compilation Buildroot (version 20071216). Ces modifications permettent de construire une toolchain fonctionnelle à destination du MIPS de la PSP. Un noyau Linux supportant un maximum de périphériques et une image comportant quelques utilitaires de Busybox ont également été intégrés à Buildroot. L'archive complète est accessible au moyen de Mercurial par :

```
hg clone http://hg.sequanux.org/buildroot-psp
```

qui crée le répertoire **buildroot-psp**. Une fois dans le dépôt, la commande **hg log** montrera les modifications apportées à l'arborescence **buildroot-20071216**. Le fichier de configuration proposé, **buildroot-20071216-linuxonpsp.config**, est renommé en **.config** avant le **make oldconfig && make**. Buildroot ira alors chercher toutes les archives nécessaires à la compilation. Comme pour toute compilation de noyau, les bibliothèques de développement **ncurses-dev** sont nécessaires.

Notre contribution dans ce projet est la réalisation d'un environnement de développement cohérent (Fig. 2) issu de Buildroot, utilisant un même ensemble d'outils (toolchain) pour compiler le noyau et les applications en espace utilisateur (notamment Busybox).

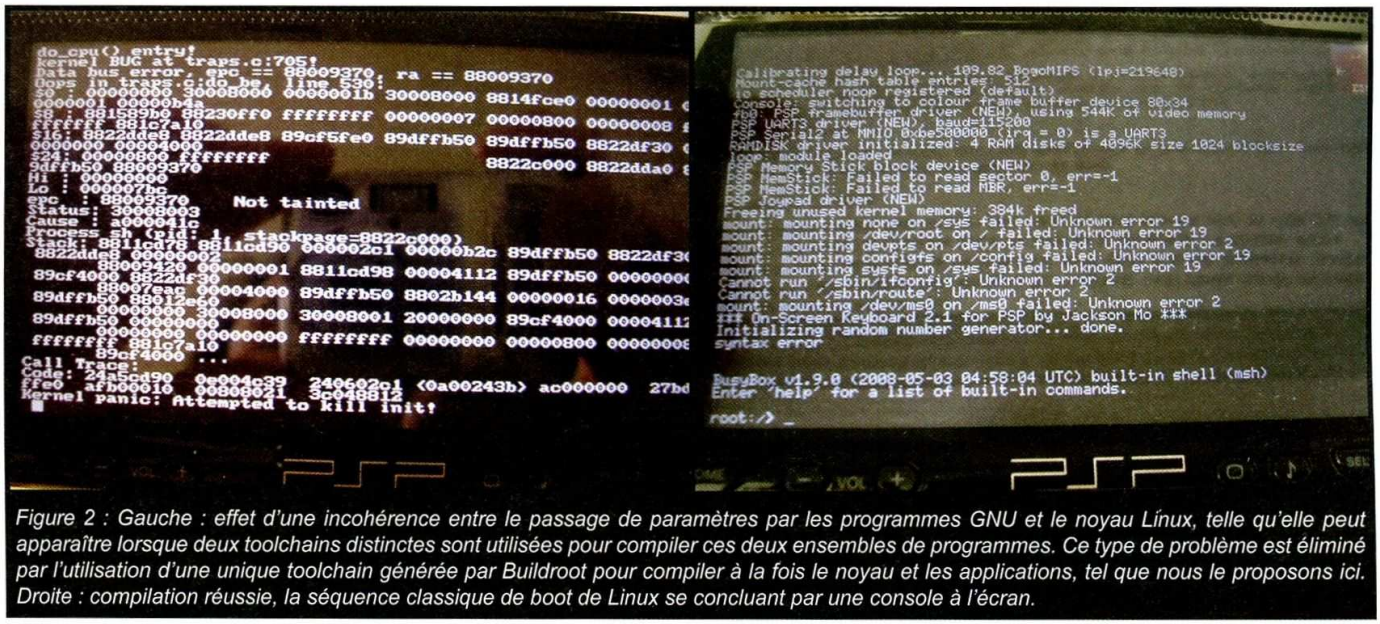


Figure 2 : Gauche : effet d'une incohérence entre le passage de paramètres par les programmes GNU et le noyau Linux, telle qu'elle peut apparaître lorsque deux toolchains distinctes sont utilisées pour compiler ces deux ensembles de programmes. Ce type de problème est éliminé par l'utilisation d'une unique toolchain générée par Buildroot pour compiler à la fois le noyau et les applications, tel que nous le proposons ici. Droite : compilation réussie, la séquence classique de boot de Linux se concluant par une console à l'écran.

1.3 Description et mise en œuvre de Buildroot

Buildroot est un ensemble de *makefiles* qui permet de générer une toolchain (outils pour compiler des programmes pour une cible donnée), un noyau Linux et un *rootfs* (image de système d'exploitation contenant fichiers de configuration et exécutables). Buildroot permet d'automatiser le téléchargement des sources et des correctifs (patch), la configuration, la compilation et l'installation des programmes dans le rootfs. Différents formats d'image rootfs sont possibles : squashfs, cramfs, ext3, archive cpio, initramfs, etc. Dans le cas de la PSP, le fichier chargé en mémoire est obtenu par compression avec gzip de l'image binaire raw **buildroot-psp/project_build_mipsel/linuxonpsp/linux-2.6.22/arch/mips/boot/vmlinux.bin**). Cette image contient un noyau et le rootfs embarqué sous la forme d'un **initramfs**. L'option **BR2_TARGET_ROOTFS_INITRAMFS=y** devra donc être sélectionnée lors de la configuration du Buildroot.

Étant donné que l'arborescence de Buildroot peut sembler au premier abord assez déroutante, nous allons en présenter les principaux répertoires et expliquer sommairement leur rôle :

- Le répertoire toolchain contient les makefiles nécessaires pour compiler la toolchain. Par exemple, **toolchain/gcc** contient de quoi construire **gcc** pour la cible visée, dans toutes ses configurations supportées :

```
$ tree -L 1 toolchain/gcc/
  toolchain/gcc/
  |-- 3.3.5
  |-- 3.3.6
…
  |-- 4.2.1
  |-- Config.in
…
  |-- gcc-uclibc-3.x.mk
  |-- gcc-uclibc-4.x.mk
```

Les fichiers **gcc-uclibc-3.x.mk et gcc-uclibc-4.x.mk** sont les fragments de makefile qui seront utilisés pour compiler **gcc** en fonction de la configuration sélectionnée.

- Lors de la construction de la toolchain, les différents composants seront compilés sous le répertoire **toolchain_build_xxxx** : pour la PSP, **xxxx** vaut **mipsel**.
- Package contient les makefiles des applications. Par exemple, **wget** (non Busybox) se trouve dans :

```
$~/buildroot$ tree package/wget/
  package/wget/
  |-- Config.in
  `-- wget.mk
```

wget.mk est le fragment de makefile permettant de compiler **wget**. Un rapide coup d'œil à ce fichier nous présente les cibles intéressantes :

```
$~buildroot/package/wget$ cat wget.mk
…
wget: uclibc $(TARGET_DIR)/$(WGET_TARGET_BINARY)

wget-clean:
 rm -f $(TARGET_DIR)/$(WGET_TARGET_BINARY)
 -$(MAKE) -C $(WGET_DIR) clean
wget-dirclean:
 rm -rf $(WGET_DIR)
…
```

À la racine du Buildroot, la commande **make wget** permet d'ajouter l'application **wget** au rootfs. **make wget-clean** permet de nettoyer le répertoire de compilation de **wget** (c'est-à-dire **build_mipsel/wget**). **make wget-dirclean** est la cible utilisée pour supprimer ce même répertoire. En règle générale, les cibles **$(app)**, **$(app)-clean** et **$(app)-dirclean** sont présentes pour toutes les applications.

La même étude sur le répertoire de Busybox présente une arborescence incluant toutes les versions successives et les correctifs associés.

- À quelques exceptions près, les applications (ou *packages*) sont compilées sous **build_mipsel** (variable **BUILD_DIR** dans les différents makefiles de paquets). Il s'agit d'un répertoire créé dynamiquement lors de la compilation. Chaque paquet y est décompressé, patché, configuré et compilé.

```
$~/buildroot$ ls build_mipsel/
fakeroot-1.8.10  makedevs  psposk2  staging_dir
```

Pour la PSP par exemple, le répertoire **build_mipsel/psposk2** est utilisé pour cross-compiler l'application On-Screen Keyboard. On notera également la présence du répertoire **build_mipsel/staging_dir**. Il est désigné dans les différents makefiles par la variable **STAGING_DIR** et est en quelque sorte la racine de l'environnement de cross-compilation. Les variables **CFLAGS** et **LDFLAGS** telles que définies par Buildroot vont respectivement aller pointer (entre autres) vers **$(STAGING_DIR)/usr/include** et **$(STAGING_DIR)/usr/lib**. Lorsqu'une bibliothèque est compilée, elle n'est pas immédiatement intégrée au rootfs. Elle est tout d'abord installée dans le répertoire **STAGING_DIR**. C'est une étape importante, surtout si une application à compiler ultérieurement dépend de cette bibliothèque. Le script **configure** de cette application aura sans doute besoin de détecter la bibliothèque ainsi que les fichiers d'en-têtes afin d'activer correctement les fonctionnalités et les options avant la compilation. Dans le cas d'une dépendance critique non résolue, la configuration ne sera tout simplement pas possible. Une bonne pratique lors du packaging d'une application est d'utiliser systématiquement le répertoire **STAGING_DIR** pour installer une application (**make install**). Les fichiers voulus sont ensuite recopiés du **STAGING_DIR** vers le rootfs. Cette étape fait office de filtre et permet par exemple d'écarter les fichiers de documentation ou alors les programmes qui ne seront pas utilisés sur le système cible. Elle permet aussi de modifier les permissions de certains fichiers.

- Les applications principales comme Busybox ou le noyau Linux ne sont pas compilées sous le répertoire **BUILD_DIR**, mais sous **project_build_mipsel/linuxonpsp** (variable **PROJECT_BUILD_DIR**) :

```
$~/buildroot$ ls project_build_mipsel/linuxonpsp/
buildroot-config  busybox-1.9.0  linux-2.6.22  linux-2.6.22-bk  root
```

On remarquera le répertoire **root**. Il est désigné dans les makefiles par la variable **TARGET_DIR**. Comme son nom le laisse deviner, il s'agit du répertoire cible rootfs qui sera utilisé pour générer une image finale au format souhaité (ext3, cpio, squashfs, cramfs, etc.). Explorer ce répertoire permet d'observer la composition de l'espace utilisateur du système cible :

```
$~/buildroot$ tree project_build_mipsel/linuxonpsp/root/
|-- bin
|   |-- busybox
|   |-- cat -> busybox
|   |-- cp -> busybox
[…]
|-- etc
|   |-- TZ
|   |-- fstab
|   |-- group
|   |-- hostname
|   |-- hosts
|   |-- init.d
|   |   |-- S20urandom
[…]
|   |-- services
|   `-- shadow
|-- home
|   `-- default
|-- init -> sbin/init
|-- lib
|   |-- libgcc_s.so -> libgcc_s.so.1
|   `-- libgcc_s.so.1
|-- linuxrc -> bin/busybox
|-- proc
|-- root
|-- sbin
|   |-- getty -> ../bin/busybox
|   |-- init -> ../bin/busybox
|   `-- mdev -> ../bin/busybox
[…]
```

Ce répertoire peut également être utilisé à des fins de développement ou de débogage. En effet, il constitue un excellent NFS root. Un des avantages d'accéder à un rootfs au travers d'un partage NFS est de rendre une application (re)compilée directement disponible sur le système cible sans avoir à reflasher ce dernier. Dans le cas de la PSP, l'absence d'interface réseau facilement exploitable empêche l'utilisation de cette technique. La PSP ne dispose que d'un contrôleur wifi et aucun pilote pour l'instant ne permet de l'exploiter. De plus, les interfaces wifi sont de très mauvaises candidates pour accéder à un root filesystem via NFS, car elles nécessitent souvent la collaboration d'un programme utilisateur (comme **wpa_supplicant**) afin d'initialiser la connexion réseau.

- Le répertoire **target** contient tous les makefiles nécessaires à la compilation du rootfs final. Par exemple, lors de la génération d'une image au format squashfs, le fragment de makefile **target/squashfs/squashfsroot.mk** sera utilisé. Nous y trouvons aussi le squelette du répertoire **TARGET_DIR** (**target skeleton**) :

```
$~/buildroot$ ls target/generic/target_skeleton/
bin  dev  etc  home  lib  mnt  opt  proc  root  sbin  tmp  usr  var
```

L'utilisateur peut utiliser ce répertoire pour inclure dans son système de fichiers un script qui n'est pas associé à une application particulière.

Le fichier **target/generic/mini_device_table.txt** contient lui la définition des fichiers spéciaux statiques du *root filesystem* final. Les permissions de certains fichiers sensibles (**/etc/passwd** ou **/etc/shadow** par exemple) y sont également définies. Lors de la création de l'image finale, le programme **makedev** se basera sur ce fichier **device_table.txt**. La compilation ne disposant bien sûr pas des permissions root, la créations des fichiers spéciaux est rendue possible en utilisant **makedev** en combinaison avec le programme **fakeroot**.

```
$~/buildroot$ cat target/generic/mini_device_table.txt
…
/etc/shadow         f     600   0     0     -     -     -     -     -
/etc/passwd         f     644   0     0     -     -     -     -     -
…
# Sony Memory stick
/dev/ms0            b     640   0     0     31    200   -     --
```

/dev/ms0 est le fichier spécial bloc qui permet d'accéder au memory stick Sony. L'utilisateur peut modifier ce fichier à sa convenance pour ajouter d'autres nœuds (*nodes* sous **/dev**).

- Comme pour le noyau Linux, le fichier de configuration se trouve sous la racine du Buildroot et porte le nom de **.config**. Il se modifie par **make config**, **make menuconfig**, etc.

En résumé, le Buildroot se divise en 2 grandes familles de répertoires : ceux contenant les makefiles, *packages* et toolchains, et les répertoires de travail créés à la compilation.

Pour finir, on soulignera un des défauts de Buildroot. Il s'agit d'une dépendance assez forte avec l'environnement *host* de cross-compilation. Lors de la configuration des applications, des outils comme **pkg-config** ont la fâcheuse tendance en cas de recherche infructueuse d'aller examiner le répertoire host **/usr/lib/pkgconfig/**. Dès lors, on imagine que la génération d'un rootfs sur un host où par exemple **dbus** est installé peut poser un certain nombre de problèmes cocasses : au mieux, une erreur de compilation et, au pire, un comportement suspect à l'exécution… Afin de parer à ce genre de désagréments, une bonne pratique est de cross-compiler au sein d'un environnement *chroot* minimaliste et maîtrisé. Une installation *debootstrap* d'une Debian stable (ou même instable) fournit un environnement host de cross-compilation très décent. Ce genre de procédure est décrit à : http://wiki.easyneuf.org/index.php/Buildroot_HOWTO.

2 Le noyau Linux pour PSP

Nous allons maintenant présenter quelques-uns des composants du noyau Linux pour la console PSP.

2.1 Le pilote memory stick Sony

Le pilote **ms_psp** permet d'accéder aux memory sticks Sony ProDuo. Les memory sticks standards ne sont eux pas supportés.

L'accès bas niveau au memory stick Sony est fourni par l'IPL SDK. Les fichiers sources concernés sont **arch/mips/psp/ipl_sdk/memstk.c** et **include/asm-mips/ipl_sdk/memstk.h**. Les fonctions exportées sont :

- **int pspMsInit(void)** ;
- **int pspMsReadSector(int sector, void *addr)** ;
- **int pspMsWriteSector(int sector, void *addr)**.

La partie du pilote s'interfaçant avec la couche bloc du noyau Linux se trouve dans le fichier **drivers/block/ms_psp.c**. La structure de ce pilote bloc **[9]** est plutôt classique.

On notera cependant quelques particularités : Le pilote **ms_psp** ne définit pas la méthode **request_fn**. De plus, il implémente sa propre méthode **make_request_fn**. Habituellement, les pilotes blocs implémentent la méthode **request_fn** pour réaliser les opérations d'entrées/sorties demandées et ne définissent pas la méthode **make_request_fn**. La fonction standard **__make_request()** mise à disposition par le module **block_core** est alors utilisée par défaut. Le rôle de cette fonction est d'organiser la file d'attente des requêtes avant de la soumettre au pilote via la méthode **request_fn**. L'intérêt de classer les requêtes est évident. Prenons l'exemple d'un périphérique de type disque dur. Les déplacements de la tête sur le disque doivent impérativement être optimisés et ce travail est celui de la fonction **__make_request()**. Cette dernière insère les nouvelles requêtes dans la file de requêtes du pilote de façon à optimiser les accès au médium et la rapidité des transferts. Pour cela, les requêtes concernant des secteurs mémoire contigus pourront être concaténées. Le tri des requêtes proprement dit est réalisé par un ordonnanceur (ou encore élévateur) bloc. Les élévateurs proposés par le noyau Linux sont : *noop*, *CFQ* (*Complete Fairness Queueing*), *deadline* ou encore *anticipatory*. Le lecteur curieux pourra trouver des informations en se référant à la documentation du noyau (**Documentation/block/**) ou encore en consultant **[10, chap. 14 "Block Device Drivers"]**. Dans le cas du memory stick Sony, dont la mémoire est de type Flash, la problématique du temps d'accès à un secteur est moins cruciale que pour un disque dur. C'est pour cela que le pilote **ms_psp** court-circuite l'ordonnanceur bloc en implémentant sa propre méthode **make_request_fn** : **psp_ms_make_request()**. Cette fonction ne classe pas et n'insère pas les BIO (blocs d'entrées/sortie) dans la file d'attente de requêtes, mais transfère directement les données. Cette implémentation évite la gestion coûteuse d'une queue de requêtes. Cette stratégie a cependant un inconvénient. Ne pas organiser les accès au médium en fonction de l'adresse des secteurs demandés ne permet pas de grouper les accès sur plusieurs secteurs mémoire contigus du memory stick. Par exemple, la lecture d'un fichier peut nécessiter l'accès à plusieurs secteurs contigus sur le périphérique. Dans ce cas, l'implémentation du **ms_psp** est telle que les secteurs seront lus un à un. Autant de commandes que de secteurs demandés seront transmises au memory stick.

```
static int psp_ms_make_request(request_queue_t * queue_, struct bio * bio_)
{
bio_endio( bio_,
     bio_->bi_size,
     psp_ms_transfer_bio( (psp_ms_partition_t *)queue_->queuedata,
     bio_ )
);
return 0;
}

static int psp_ms_transfer_bio
(
  psp_ms_partition_t * partition_,
  struct bio * bio_
)
{
  struct bio_vec * bvec;
  int i, rt;
  void * buf;
  sector_t sector, numSectors;
  /* Calculate the start sector */
  sector = bio_->bi_sector + partition_->startSector;

  bio_for_each_segment( bvec, bio_, i )
  {
    buf = __bio_kmap_atomic( bio_, i, KM_USER0 );
    //numSectors = bio_cur_sectors( bio_ );
    numSectors = bvec->bv_len >> 9;

    if ( bio_data_dir( bio_ ) == READ )
    {
      rt = psp_ms_read( buf, sector, numSectors );
    }
    else
    {
      rt = psp_ms_write( buf, sector, numSectors );
    }
    /* always call unmap regardless of the operation succeeded or not */
    __bio_kunmap_atomic( bio_, KM_USER0 );

    if ( rt < 0 )
    {
      DBG(( "Failed to %s MS (%d,%d) with error of %d\n",
           ( bio_data_dir( bio_ ) == READ ) ? "read" : "write",
            sector, numSectors, rt ));
```

```
      return -EIO;
    }

    sector += numSectors;
  }

  return 0;
}
```

Les structures bio transmises à la fonction **psp_ms_make_request()**, puis à la fonction **psp_ms_transfer_bio()** ne seront composées que d'un seul segment (**struct bio_vec**). En effet, la fusion de requêtes bio adjacentes normalement effectuées par l'élévateur bloc n'est pas implémentée par le pilote **ms_psp**. La macro **bio_for_each_segment()** ne produira donc qu'une seule itération. Étant donné qu'un segment est d'au mieux de la taille d'une page (4096 octets), la fonction **psp_ms_read()** sera donc appelée pour transmettre au maximum 8 secteurs (de 512 octets).

```
static int psp_ms_read(void * buf_, int sector_, int numSectors_)
{
  int t, i, rt;
  char * readBuf;

  if ( down_interruptible( &s_psp_ms_rw_sem ) != 0 )
  {
    return -EBUSY;
  }

  for ( readBuf = (char *)buf_, i = 0; i < numSectors_; i++ )
  {
    rt = psp_ms_read_sector( readBuf, sector_ + i );
    if ( rt < 0 )
    {
      DBG(( "%s: Failed to read sector %d, err=%d\n",
            PSP_MS_NAME, sector_ + i, rt ));
      break;
    }

    readBuf += PSP_MS_SECTOR_SIZE;
  }

  up( &s_psp_ms_rw_sem );
  return rt;
}
```

De plus, l'IPL ne permet de traiter qu'un secteur à la fois. La fonction **psp_ms_read_sector()** sera donc appelée pour chacun des secteurs à lire.

Assurément, le pilote **ms_psp** et l'IPL permettent de dialoguer avec un memory stick, mais des optimisations sont possibles :

- ajouter à l'IPL le support pour transférer les secteurs contigus du memory stick via une seule commande ;
- utiliser un élévateur bloc afin d'agglomérer les requêtes concernant des secteurs adjacents.

Un pilote *memstick* a récemment été intégré au noyau Linux. Son implémentation est bien plus optimisée et complète que celle du pilote **ms_psp**. Il supporte par exemple les memory sticks standards. Une amélioration intéressante serait donc d'intégrer ce pilote au noyau Linux PSP et d'y ajouter un pilote host PSP en regroupant et complétant les fonctions bas niveau exportées par l'IPL. Pour l'instant, les contrôleurs hosts supportés par le pilote memstick sont les lecteurs de cartes TI Flash Media et JMicron JMB38X.

2.2 Le pilote du joypad

Le noyau Linux pour PSP fournit un pilote (**psp_joypad**) permettant d'utiliser le *joypad* et les boutons de la console. Le code source de ce pilote est localisé dans le fichier **drivers/input/joypad_psp.c**. Contrairement à ce que le chemin (*path*) peut laisser penser, ce pilote n'utilise malheureusement pas la couche *input* du noyau. Il s'agit en fait d'un pilote de type caractère classique.

La connaissance et le support de l'architecture de la PSP par le noyau Linux ne permettent pas encore d'associer une ligne d'interruption aux mouvements du joypad ou à l'appui d'un bouton. Le pilote a donc recours au *polling* : un *thread kernel* dédié collecte à intervalle régulier la valeur du registre de statut des boutons et du joypad. L'accès bas niveau à ce registre est fourni une fois de plus par l'IPL SDK via la fonction **_pspSysconGetCtrl1()**.

```
static int __init psp_joypad_init()
{
  [...]

  INIT_LIST_HEAD( &s_psp_joypad_queue_list );

  /* Launch the daemon thread here */
  s_psp_joypad_thread_id = kernel_thread( psp_joypad_thread,
                                          NULL,
                                          CLONE_FS | CLONE_SIGHAND );

  [...]
}

static int psp_joypad_thread(void * unused_)
{
  unsigned long val;
  struct list_head * pos;

  while ( !s_psp_joypad_thread_terminated )
  {
    if ( psp_joypad_read_input( &val ) )
    {
      DBG(( "%s: %08x\n", PSP_JOYPAD_NAME, val ));

      /* Turn the LCD back on whenever a button is pressed */
      psp_lcd_on();

      /* Feed the data to all registered queues */
      if ( down_interruptible( &s_psp_joypad_queue_list_sem ) == 0 )
      {
        list_for_each( pos, &s_psp_joypad_queue_list )
        {
          (void)psp_joypad_queue_push( LIST_TO_QUEUE( pos ), val );
        }
        up( &s_psp_joypad_queue_list_sem );

        wake_up_interruptible( &s_psp_joypad_wait_queue );
      }
    } // end if

    msleep( 1000 / PSP_JOYPAD_SAMPLE_RATE );
  } // end while

  return 0;
}
```

La fonction **psp_joypad_thread()** est donc le cœur du pilote **psp_joypad**. Elle accomplit les tâches suivantes :

1. Collecter périodiquement (toutes les 50 millisecondes) la valeur du registre de **status** du joypad en appelant la fonction **psp_joypad_read_input()**.

2. Ajouter les valeurs recueillies dans une file d'attente type FIFO.

3. Et enfin réveiller les processus en attente de lecture.

Un processus souhaitant connaître l'état des boutons et du joypad peut par exemple utiliser la méthode **read()** exportée par le pilote **psp_joypad**. Pour cela, il lui suffit d'ouvrir le fichier spécial **/dev/joypad** (majeur 11, mineur 200) et de d'utiliser l'appel système **read()**. Ce dernier conduit le fil d'exécution du processus jusqu'à la méthode **psp_joypad_fop_read()** (exportée par le pilote). Le processus est ensuite inscrit dans la file d'attente de lecture du pilote. Il sera réveillé lorsque des données seront disponibles. Un exemple d'utilisation de cette interface pourra être étudié en consultant le code source du PSP OSK (PSP *On Screen Keyboard*).

Le pilote **psp_joypad** fonctionne relativement bien, mais des améliorations sont possibles.

Tout d'abord, le pilote pourrait et devrait utiliser le module input proposé par le noyau Linux. Ce module offre un certain nombre de facilités et permet notamment d'exporter une interface standard vers l'espace utilisateur. Par exemple, le joypad et les boutons de la PSP deviendraient directement utilisables pour toutes les applications supportant l'interface evdev. Ce qui est le cas du serveur X. La documentation du noyau (**Documentation/input/input-programming.txt**) présente un exemple simple d'implémentation pour un *input device driver*. Cet exemple pourrait constituer un excellent squelette pour une nouvelle version du pilote **psp_joypad**.

Une autre amélioration possible serait évidemment d'identifier et d'utiliser la ligne d'interruption associée au joypad et aux boutons de la PSP.

2.3 Le pilote série

La PSP est munie de plusieurs ports série : soit la liaison filaire par le connecteur à côté du jack pour le casque, soit la liaison infrarouge. Bien qu'une interruption soit associée aux évènements survenus sur les ports de communication asynchrones (UART défini comme l'interruption 0 **[7]**), l'implémentation proposée ici exploite – probablement pour des raisons historiques – la communication en sondant périodiquement l'état de ces ports, en même temps que la gestion d'autres évènements sous le contrôle de l'interruption *timer* (interruption **CPUTIMER** numéro 66 dans **[7]**).

La communication par port série UART3 a été implémentée dans **linux/drivers/serial/serial_psp.c** :

```
void psp_uart3_txrx_tick()
{
  if ( !s_psp_uart3_port_data.shutdown &&
       ( s_psp_uart3_port_data.txStarted ||
         ( !s_psp_uart3_port_data.rxStopped &&
           !( PSP_UART3_STATUS & PSP_UART3_MASK_RXEMPTY ) ) ) )
  {
    up( &s_psp_uart3_port_data.sem );  // reveille la reception du char
  }
}
```

Ce gestionnaire d'interruption réveille un thread noyau chargé de tester la présence d'un caractère dans la file de réception de l'UART (Fig. 3) : **psp_port_txrx_thread()**. Le réveil se fait donc par déclenchement d'une interruption timer, initialisée dans **psp.c** sous **arch/mips/psp** (fonction **psp_cputimer_handler()**). Lors de la lecture des données, l'accès au port est bloqué par un *mutex* **init_MUTEX_LOCKED(&portData->sem);** qui est initialisé par le thread noyau **psp_port_txrx_thread()**.

On retrouve ici une initialisation telle que celle présentée plus haut pour le joypad :

```
static int __init psp_serial_modinit()
{
[…]
portData->txThreadId = kernel_thread( psp_port_txrx_thread,
                                      port,
                                      CLONE_FS | CLONE_SIGHAND );
[…]}
```

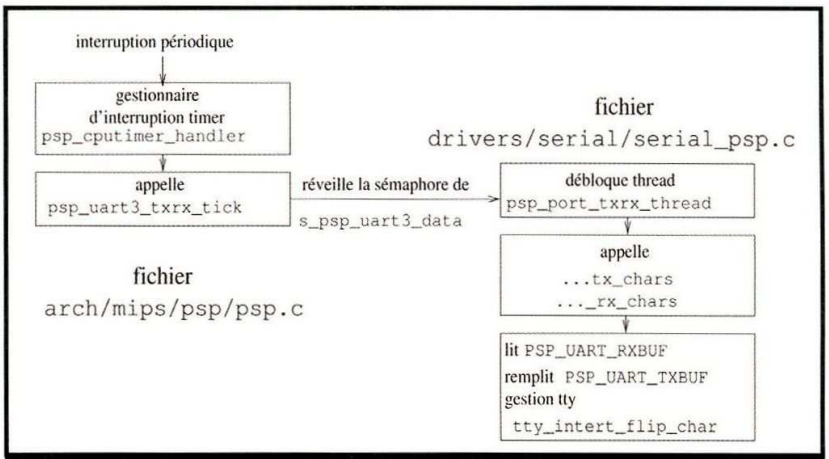


Figure 3 : Interaction entre le gestionnaire d'interruption timer et de communication avec les ports série pour périodiquement sonder l'état de ces ports et interagir avec la console.

La gestion des ports série de la PSP, nommés **/dev/ttySRCi**, est décrite dans le fichier **serial_psp.c** du sous-répertoire **drivers/serial** de l'arborescence du noyau. Nous y trouverons d'une part une gestion du protocole de communication fortement inspirée de la description de la configuration de l'UART proposée dans l'IPL SDK et dans les logiciels dédiés à la PSP, et, d'autre part, l'interfaçage du port série avec un terminal – et notamment une console – probablement l'originalité la plus intéressante de ce pilote.

3 Exécution et console série

Étant donné que le bootloader (le jeu exécuté en natif sur l'OS Sony de la PSP) se charge d'initialiser les périphériques en exploitant les fonctions disponibles initialement sur la console, Linux n'a pas besoin de se charger de cette tâche : on ne trouvera donc pas de fonction boot comme c'est habituellement le cas dans **linux-2.6.22/arch/**, mais seulement un appel aux fonctions spécifiques à la PSP lors du démarrage du kernel par **arch/mips/kernel**. Néanmoins, les fonctions dédiées à l'architecture de la console se retrouvent bien dans **linux-2.6.22/arch/mips/psp**, et notamment la gestion des interruptions et la réinitialisation périodique du chien de garde *watchdog*.

3.1 Ajout d'un clavier

L'injection de caractères dans la couche tty **[8, chap. 18 "TTY Drivers"]** par le gestionnaire de port série (fonction **tty_flip_char()** de **psp_uart3_rx_chars()**) s'exploite de deux façons : soit en connectant un PC sur lequel tourne un logiciel de communication sur le port série (*screen* ou *minicom* par exemple), soit en connectant un clavier de PC dont le protocole de communication a été transformé en RS232. La première solution est la plus simple, mais la moins pratique puisqu'elle nécessite un PC, réduisant quelque peu l'intérêt d'une PSP sous GNU/Linux. La seconde solution est abordée en détail ici.

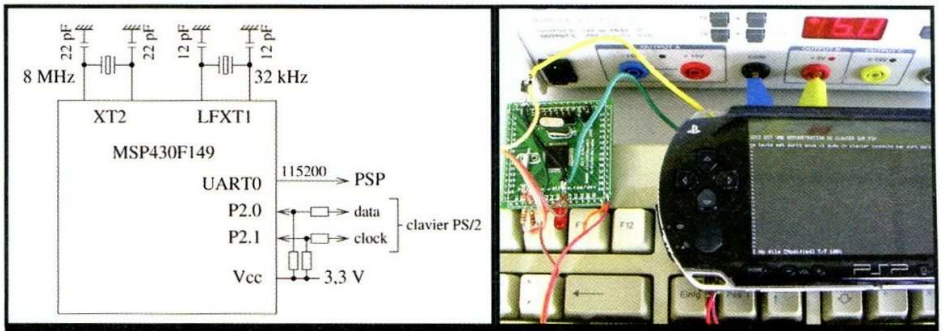


Figure 5 : Circuit de conversion entre un clavier PS2 et un format RS232 compatible avec la PSP. Un microcontrôleur est dédié à cette tâche : le protocole synchrone bidirectionnel de communication avec les périphériques PS2 y est implémenté de façon logicielle. Seul un caractère sur deux est transmis à la PSP, puisqu'un clavier transmet le code de touche à chaque appui et relâchement de la touche concernée. Sur la figure de droite, vi est fonctionnel sur PSP : la console devient ainsi un outil de travail autonome ne nécessitant pas un ordinateur additionnel pour communiquer.

Cette fonctionnalité entre évidemment en conflit avec la disponibilité d'un clavier sur le port série : il est donc dans ce cas nécessaire de désactiver la réception des caractères pour injection dans la couche clavier. Réciproquement, l'activation du clavier sur le port série nécessite de désactiver la console qui y est associée pour éviter tout conflit.

Nous avons ajouté la communication avec un clavier PS2 par la voie la plus simple : conversion de PS2 (protocole synchrone bidirectionnel) en RS232 (protocole asynchrone avec des voies séparées pour l'émission et la réception de données) au moyen d'un MSP430. Le travail est minimal, puisqu'un exemple quasiment fonctionnel est fourni avec **msp-gcc** (version de **gcc** pour MSP430) dans **examples/mspgcc/pc_keyboard**. Sans entrer dans les détails du protocole PS2 qui est parfaitement géré par le microcontrôleur, nous nous contentons de modifier ce programme pour communiquer par le port série au lieu d'afficher les caractères sur un écran, et de retirer un code clavier sur deux puisque appuyer et relâcher une touche induisent tous deux la transmission du code correspondant. Le **main()** du programme flashé dans le MSP430 devient donc pour notre application :

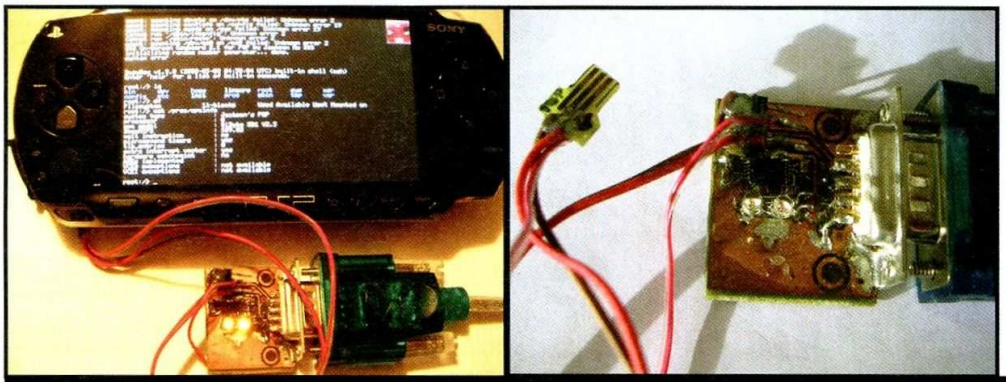


Figure 4 : Circuit de conversion RS232-connecteur série de la PSP, à côté du connecteur jack audio. Il s'agit simplement d'un MAX3232 ou équivalent transformant les niveaux ±12 V en 0-3,3 V. Ce montage permet de communiquer au moyen de minicom par exemple avec un système GNU/Linux sur PSP lorsqu'une console est connectée au port série /dev/ttyS2..

Du point de vue utilisateur, la création d'une console sur le port série (Fig. 4) s'obtient par l'ajout dans **/etc/fstab** de la ligne :

```
# Put a getty on the serial port
ttyS2::respawn:/sbin/getty -L ttyS2 115200 vt100
```

```
while (1) {                          //main loop, never ends…
    LPM0;                            //sync, wakeup by irq
    processRawq();
    while( (c = getkey()) ) {
        if ((c >> 8) == 0) {
            if (c=='\n') {putc(10);} // putc(13);}
             else
```

```
            if (makebreak==0) {putc(c);makebreak=1;}
               else makebreak=0;
        } else {
            switch (c) {
                case CAPS:
                    sendkeyb(0xed);
                    delay(1000);
                    sendkeyb((s & 1) ? (BIT0|BIT1) : 0);
                    break;
                case F1:
                   putc('h'); putc('e'); putc('l'); putc('l'); putc('o');
            }
        }
    }
}
```

Ces tâches très simples peuvent être exécutées par n'importe quel microcontrôleur : nous avons, pour notre part, utilisé un MSP430F149 (Fig. 4) à ces fins. Le programme se compile au moyen de **msp430-gcc** et des *binutils* associés que nous avions décrits par ailleurs **[11]**.

3.2 Utilisation des boutons

En complément du clavier relié au port série – ou pour les moins électroniciens qui ne désirent pas implémenter le montage présenté précédemment – une solution purement logicielle est fournie sous la forme du PSP OSK (On Screen Keyboard). Il s'agit d'une interface dédiée à la PSP qui affiche quelques touches sur l'écran, le choix de la touche étant effectué par combinaison des boutons de contrôle de la PSP. Sans permettre de taper un long texte (les amateurs de SMS me contrediront peut-être), cette interface permet au moins de visualiser le contenu de quelques fichiers ou tester quelques fonctionnalités nouvellement compilées dans Busybox (Fig. 6).

Figure 6 : L'OSK est visible en haut à gauche de l'écran de ces photos de la PSP. Il s'agit du carré rouge contenant les symboles de lettres dont le choix se fait par des combinaisons des boutons de la PSP. Bien que d'un accès peu confortable, il permet de valider la compilation du système avec des commandes simples telles que dmesg, uname -a (gauche) ou cat /proc/cpuinfo (droite).

4 Développements additionnels

Nous nous sommes contentés dans cette présentation d'installer un noyau Linux et quelques outils additionnels afin de rendre la console exploitable. La gestion de l'écran graphique est implémentée sous forme de *framebuffer* (**drivers/video/pspfb.c** dans l'arborescence du noyau) et donne donc accès à tous les programmes qui supportent ce périphérique (incluant notamment ceux compilés avec la bibliothèque SDL ou au moyen de **qtopia**, la version embarquée de **Qt**).

Le matériel inclus dans la PSP est en grande partie propriétaire et non documenté. Les applications fonctionnant sous PSP font appel aux fonctions fournies par le système d'exploitation Sony : il s'agit des nombreux appels aux fonctions dont les noms commencent par **sce**, référencées dans le PSP SDK disponible à http://ps2dev.org/psp/Projects. Nous ne savons pas ce que font ces fonctions : nous nous contentons de les appeler aveuglément et d'en attendre une réponse. Cette solution n'est pas exploitable par GNU/Linux exécuté en remplacement de l'OS Sony, puisque, dans ce cas, les fonctions ne sont plus utilisables (l'espace RAM contenant l'OS Sony, et donc ces fameuses fonctions **sce***, ont été effacés pour laisser la place à Linux).

Une solution consiste alors à profiter du travail des contributeurs à IPL qui désassemblent certaines fonctions fondamentales de l'OS Sony et fournissent sous forme de code source des fonctions effectuant les mêmes opérations. L'archive de ces codes source est disponible à http://www.mediafire.com/?etdnoonpnfj. Dans

l'arborescence de développement du noyau Linux pour PSP, ces fonctions se retrouvent dans `linux-2.6.22/include/asm-mips/ipl_sdk` pour les prototypes, et `linux-2.6.22/arch/mips/psp/ipl_sdk`. Ainsi, seule l'étude des codes désassemblés du système d'exploitation natif de Sony permettra de découvrir le fonctionnement de périphériques aussi complexes que les interfaces wifi ou USB. Un effort en ce sens a déjà permis l'utilisation des Memory Stick Pro depuis uClinux grâce aux outils fournis dans le IPL SDK http://exophase.com/psp/booster-releases-psp-ipl-sdk-2190.htm.

5 Conclusion

Nous avons présenté l'exploitation d'une console de jeu, la Playstation Portable, comme environnement de développement pour uClinux sur une architecture intéressante à base de processeur MIPS. Cette activité a introduit un certain nombre de tâches qui se retrouvent sur tout port d'un système d'exploitation sur une nouvelle architecture embarquée :

- Écrire un bootloader qui initialise les périphériques du processeur et charge le système d'exploitation en mémoire.
- Modifier le système d'exploitation afin de l'adapter à sa propre application. Ici, l'affichage se fait sur écran graphique, tandis que la communication se fait soit au moyen des boutons de la PSP, soit au moyen d'un clavier qui s'adapte sur le port série dont est équipée la console.
- Identifier le matériel afin de développer les pilotes appropriés pour utiliser au mieux les fonctionnalités du système d'exploitation.

À notre connaissance, la majorité des périphériques ne sont pas exploitables à ce jour. Seules les MemoryStick Pro sont supportées. Le port USB et le wifi ne sont pas documentés. Il s'agit donc d'une plateforme pour laquelle les développeurs sont encore susceptibles de fournir des contributions pertinentes.

Références

- **[1]** HEITOR (É.), « *Jugamos a a Fonera* », *GNU/Linux Magazine France* 94, mai 2007.
- **[2]** Xiptech est une société qui met à disposition un certain nombre d'outils et une toolchain à destination de l'architecture MIPS : Porting uClinux to MIPS est disponible à **http://www.xiptech.com/download/porting.zip**
- **[3]** FRIEDT (J.-M.), « Introduction à la programmation sur PlayStation Portable », *GNU/Linux Magazine France* 92, mars 2007.
- **[4]** Le site web de Jackson Mo « *Linux On PSP* » regroupe les outils dont nous nous sommes servis pour générer un Buildroot fonctionnel : **jacksonm88.googlepages.com/linuxonpsp.htm**
- **[5]** Le site original **df38.dot5hosting.com/~remember/chris/** n'est plus actif : un complément qui s'en approche est **http://www.bitvis.se/articles/psplinux.php**
- **[6]** TyRaNiD, « *The Naked PSP* », présentation disponible à **http://ps2dev.org/psp/Tutorials/PSP_Seminar_from_Assembly.download**
- **[7]** GROEPAZ/HITMEN, « *Yet another PlayStationPortable Documentation* », disponible à **http://hitmen.c02.at/files/yapspd/psp_doc/**, et en particulier la liste des interruptions de la PSP à **http://hitmen.c02.at/files/yapspd/psp_doc/chap9.html**
- **[8]** CORBET (J.), RUBINI (A.) & KROAH-HARTMAN (G.), *Linux device drivers*, O'Reilly, 2005, disponble à **http://lwn.net/Kernel/LDD3**
- **[9]** PETAZZONI (T.) & DECOTIGNY (D.), Conception d'OS : pilotes de périphériques blocs, *GNU/Linux Magazine France* 80 (janvier 2006), pp. 74-90.
- **[10]** BOVET (Daniel P.) & CESATI (Marco), *Understanding the Linux kernel*, O'Reilly, novembre 2005, troisième édition.
- **[11]** FRIEDT (J.-M.), MASSE (A.), BASSIGNOT (F.), Les microcontrôleurs MSP430 pour les applications faibles consommations – asservissement d'un oscillateur sur le GPS., *GNU/Linux Magazine France* 98, octobre 2007.

Auteurs : Simon Guinot & Jean-Michel Friedt

Simon Guinot est développeur de systèmes embarqués et noyau Linux, membre de l'association Sequanux pour la diffusion du logiciel libre en Franche Comté. Il est joignable sur IRC sur **#sequanux**, serveur **irc.freenode.net**.

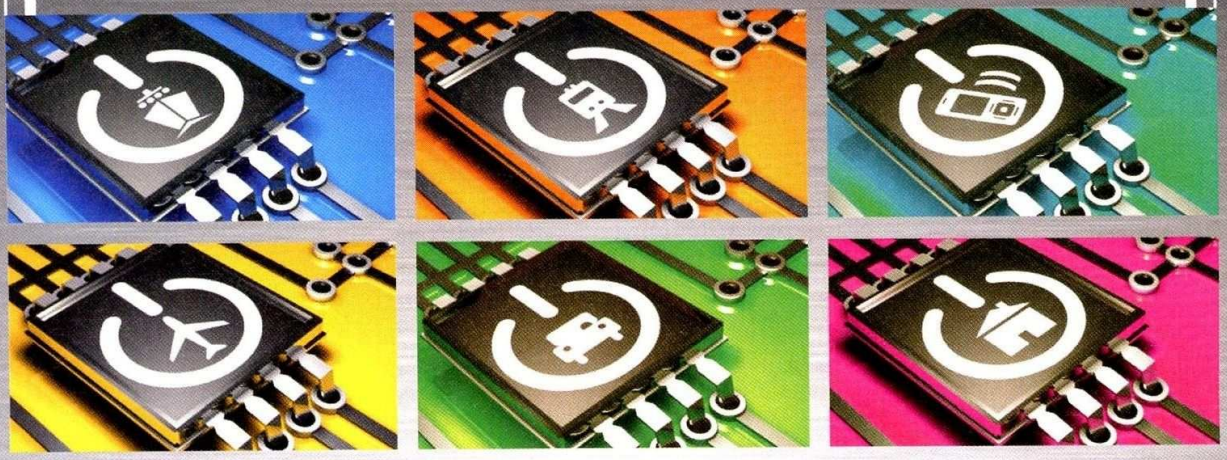
Jean-Michel Friedt est membre de l'association Projet Aurore pour la diffusion de la culture scientifique et technique à Besançon. Afin de maîtriser diverses architectures de processeurs en vue du développement de systèmes embarqués, il s'intéresse à l'exploitation d'outils libres en vue de porter des logiciels complexes (par exemple le noyau Linux) sur des plateformes aux ressources réduites. Les consoles de jeu offrent une plateforme souvent disponible ou, tout au moins, accessible à moindres coûts.

rts

17e édition

Le salon des solutions informatiques temps réel et des systèmes embarqués

EMBEDDED SYSTEMS 2009



31 mars, 1 & 2 avril 2009 - Paris Expo - Porte de Versailles

Pour exposer, visiter l'exposition ou s'inscrire aux conférences : www.groupesolutions.com

Participez à l'événement 100% Embedded et Real-Time !

Exposition : 150 exposants attendus sur un même plateau.

Technologies & matériels : ■ Cartes et composants ■ Modules ■ OS temps réel ■ Conception ■ Environnements de développement ■ Outils de test et de validation

Conférences & tables rondes, ateliers :

■ Architectures multicoeurs ■ FPGA ■ Techniques de virtualisation ■ Plates-formes de développement virtuelle ■ Environnements Open Source ■ Modules processeurs ■ Recherches appliquées en ingénierie logicielle ■ Autosar

Rts EMBEDDED SYSTEMS est organisé en parallèle à :

display 2009 — 11e Salon de l'affichage et de la visualisation électronique

&

MACHINE To MACHINE 2009 — 4e Salon des solutions MtoM

Salon strictement réservé aux professionnels.

Organisation BIRP 97, rue du Cherche-Midi 75006 Paris - France - Tél. : +33 (0)1 44 78 99 30 - Fax : +33 (0)1 44 78 99 49 - e-mail : info@salon-rts.com

DOMOGIK : commandez votre

La domotique peut être définie comme l'ensemble des moyens vous permettant d'automatiser tout ou partie de l'équipement de votre maison. On peut citer, comme exemples, la lumière, les volets, le portail, mais aussi des éléments moins « mécaniques » comme le téléphone, la musique, la météo, etc.
Si de (trop) nombreuses technologies existent sur le plan matériel, il est en revanche plus compliqué de trouver une interface logicielle permettant de centraliser la programmation, la visualisation ou la supervision des éléments automatisés au sein de la maison. C'est face à ce constat que le projet Domogik a vu le jour.

Auteur

- Maxence Dunnewind

À l'origine, un fil de discussion sur le forum d'une communauté d'une célèbre distribution GNU/Linux (Ubuntu, pour ne pas la nommer). La création de ce sujet a permis à plusieurs personnes de se rendre compte de deux choses :

- Il n'existe que peu de logiciels libres permettant de centraliser le fonctionnement des équipements domotiques d'une maison.
- Plusieurs personnes étaient intéressées par la création d'un nouveau projet, certaines ayant déjà commencé à développer un petit-quelque-chose à titre privé.

L'idée est alors venue de lancer le développement d'un nouveau projet ayant pour but de permettre la gestion de la domotique d'une habitation. Deux personnes ont alors créé le projet Domogik. Cet article, se voulant peu technique, a pour but de présenter le projet Domogik, son fonctionnement et ses fonctionnalités.

1 La structure du projet

Avant toute chose, Domogik est un logiciel libre sous licence GPLv3. Son code source, disponible publiquement, est développé par des bénévoles intéressés par le domaine de la domotique. Il utilise également d'autres projets libres. Bien entendu, vous êtes libre de venir donner un coup de main si vous le désirez. Ceci étant dit, passons aux choses « sérieuses ».

Vu la quantité d'informations pouvant être présente sur une interface de gestion domotique (tant d'un point de vue présentation à l'utilisateur que d'un point de vue gestion interne), il apparaissait nécessaire de définir une architecture propre et la plus évolutive possible. Il a donc été rapidement décidé de séparer le projet en deux parties :

- le système de gestion interne ;
- l'interface de présentation et de contrôle par l'utilisateur.

Nous allons donc détailler chacune des deux parties (sans pour autant entrer dans les détails).

1.1 Le système de gestion interne

Le système interne doit permettre d'interconnecter les différentes technologies présentes dans l'habitation et d'en gérer le fonctionnement de façon indépendante. Il doit en outre être capable de prendre en compte la configuration de base définie par l'utilisateur, mais aussi les modifications dynamiques de la configuration ou bien les ordres ponctuels. Par ailleurs, il doit être capable de stocker de façon rapide et fréquente l'état des différents équipements, ceci afin que l'utilisateur puisse les visualiser au travers de son interface.

Pour effectuer cette tâche, le système se base sur un réseau xPL. Le fonctionnement détaillé de ce protocole ne sera pas expliqué ici, car il l'a déjà été dans de précédents numéros. Une explication sommaire va cependant être effectuée, afin de comprendre son fonctionnement au sein de l'application Domogik.

maison avec des logiciels libres

xPL est un protocole permettant de définir des standards de communication de messages dans le cadre d'une utilisation domotique. Sa structure peut être (basiquement) comparée à un réseau d'autoroutes. Ainsi, chaque grande ville est « connectée » à une ou plusieurs autoroutes. Les autoroutes sont également connectées entre elles à l'aide d'échangeurs, ayant pour but de permettre aux véhicules d'une autoroute de passer sur une autre autoroute afin de joindre une ville.

On peut donc comparer les échangeurs à des « hubs » xPL, les villes à des clients du réseau et les véhicules à des messages. Ainsi, chaque message (ou véhicule) possède la même « structure » (4 roues, un moteur, etc.), mais pas nécessairement le même contenu. Un message part d'un client (une ville), passe à travers un ou plusieurs hubs (ou échangeurs) pour arriver à un autre client (ou ville).

D'un point de vue logiciel, un hub est un serveur écoutant sur un port réseau. Chaque client s'y établit une connexion avec le hub. Lorsqu'un client envoie un message, le hub le retransmet à tous les autres clients. Par ailleurs, et c'est là la partie la plus intéressante du protocole xPL, chaque message est « rédigé » suivant un schéma annoncé dans son en-tête. Ainsi, il suffit que deux clients soient capables de décoder un même schéma pour pouvoir communiquer à l'aide de ce schéma. En théorie, chaque schéma correspond à une technologie. Le schéma définit les couples clés – valeur pouvant apparaître dans le message. Un certain nombre de schémas sont définis comme « standards » par les développeurs du protocole xPL. Cependant, il est tout à fait possible de définir des schémas personnalisés.

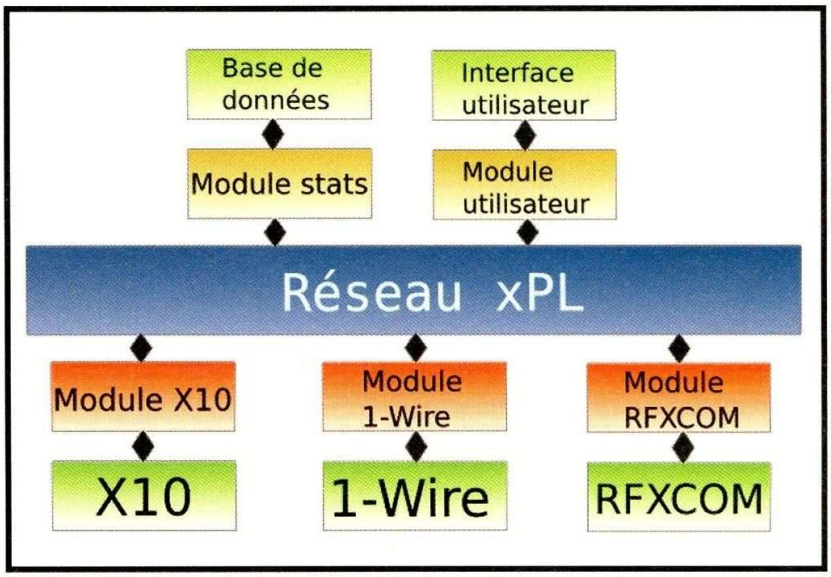
Par ailleurs, chaque client possède une identité, ce qui permet d'adresser des messages.

C'est donc sur cette technologie que se base le cœur du projet Domogik. Ainsi, le système interne est découpé en modules, chaque module effectuant une tâche spécifique, et communiquant avec le reste du réseau via le protocole xPL. Un module pourra soit servir d'interface avec une technologie, soit être dédié à une tâche (stockage des informations transmises, gestion de la configuration, etc.).

Chaque module possède un nom unique sur le réseau, permettant d'en identifier la fonction. Cependant, malgré les possibilités de gestion des destinataires du protocole xPL, il a été choisi que, sauf cas particulier, les messages envoyés par les modules seraient diffusés à tous les destinataires (*broadcast*). Ainsi, un émetteur n'a pas à savoir qui doit utiliser le message, ni comment. Il se contente de transmettre ses messages.

Ceci implique qu'un module doit être capable de savoir quels messages lui sont destinés. Ceci est défini au moment du développement des modules, le plus souvent en rapport avec la technologie gérée.

Pour résumer l'explication de l'architecture interne du système de gestion, nous pouvons représenter un début de réseau comme sur le schéma ci-dessous :



Chaque module sert donc d'interface entre un élément extérieur (technologie matérielle, base de données, interface utilisateur) et le reste du système, à travers le réseau xPL. Même si le projet Domogik est encore jeune, nous pouvons déjà lister un certain nombre de modules existant ou qui existeront dans les prochaines semaines :

- Un module par technologie supportée.
- Un module de stockage des états dans une base (suivi de l'état des lampes, des températures, etc.).
- Un module de déclenchement des actions définies de façon permanente par l'utilisateur (allumage de l'arrosage, allumage des chauffages en dessous d'une certaine température, ouverture des volets tous les matins, etc.).
- Un module de communication avec l'interface utilisateur permettant de déclencher des actions ponctuelles.

Un module peut effectuer 3 types d'opérations :

- Réalisation d'une action lors de la réception d'un message xPL (changement d'état d'une lampe, démarrage ou arrêt de la musique, etc.).
- Envoi d'un message lors d'un changement d'état ponctuel (ouverture d'une porte, allumage ou extinction d'une lumière, etc.).
- Envoi d'un message de façon périodique avec certaines informations variables (valeur des différents thermomètres, état de portes ou fenêtres, etc.).

Les messages envoyés seront reçus par tous les autres modules, chacun sachant s'il lui est destiné en fonction du schéma utilisé et potentiellement du destinataire déclaré dans le message.

Tous les modules ont été développés à l'aide du langage Python. Ceux ci se basent sur une bibliothèque elle-même développée en Python et qui permet de communiquer de façon simple avec un réseau xPL. Les bibliothèques permettant d'utiliser des technologies (matérielles ou logicielles) existent le plus souvent déjà en Python. Elles sont alors utilisées. Il arrive cependant que certaines bibliothèques soient développées spécialement pour les besoins du projet Domogik (par exemple les bibliothèques de communication via DBUS et la spécification MPRIS).

Maintenant que vous avez une idée un peu plus précise du fonctionnement interne du système de gestion de Domogik, nous allons nous intéresser à la partie visible de l'iceberg : l'interface utilisateur.

1.2 L'interface utilisateur

Un système domotique, bien qu'ayant pour but premier l'automatisation d'un certain nombre de tâches répétitives, n'a que bien peu d'intérêt si l'utilisateur n'a pas la possibilité d'interagir avec l'ensemble des équipements. C'est pourquoi le projet Domogik considère que le développement de l'interface utilisateur est une opération importante. Elle reste cependant distincte du reste du système, car ledit système doit être autonome, l'interface n'apportant qu'une fonctionnalité d'observation et de commande ponctuelle (rares seront les utilisateurs à rester 24h sur 24 devant l'interface).

Elle a donc été dissociée du reste du projet dès le début, afin de permettre une évolutivité et une indépendance maximale. À l'origine, un début de développement avait été effectué par l'un des créateurs pour son usage personnel. Ce développement, effectué en PHP, s'est bien vite avéré difficilement adaptable et, surtout, difficilement évolutif. Par ailleurs, afin d'homogénéiser les développements au sein du projet, il a été choisi d'abandonner PHP. Ainsi, l'interface utilisateur est développée en Python à l'aide du *framework* Django, qui permet de grandement faciliter le développement d'interfaces Web en Python.

Côté apparence, l'interface se découpera en 3 grandes parties. Un menu permettra à l'utilisateur de sélectionner une pièce de l'habitation, un autre lui proposera alors les différents types d'interactions possibles pour cette pièce, en fonction des éléments y étant présents (contrôle des lumières, des volets, de la musique, des équipements électriques, observation de l'évolution de la température, etc.). Une fois que l'utilisateur aura sélectionné l'interaction souhaitée, les informations correspondantes seront alors affichées sur la page. Il sera également possible de filtrer les équipements par zone, pièce ou type.

Certaines pages seront purement statiques, et d'autres se voudront dynamiques, comme le contrôle des équipements électriques ou de la musique. L'utilisateur pourra alors, via un simple clic sur l'interface (à l'aide de la souris ou d'un écran tactile), déclencher l'action souhaitée.

Un module Python permettra une communication entre l'interface utilisateur et le système xPL. Cependant, ce module devrait majoritairement être utilisé pour l'envoi d'ordres à partir de l'interface, la récupération des informations d'état se faisant de façon passive par des accès à la base de données dans laquelle seront stockés les états des équipements.

Par ailleurs, des menus supplémentaires seront présents, pour permettre notamment l'accès à des interfaces de configuration du système, afin, par exemple, de rajouter une règle de déclenchement d'une action. Cette partie de l'interface devra permettre de visualiser les règles existantes sous une forme simple, et d'ajouter des nouvelles règles simples ou complexes. On peut donner quelques exemples du genre d'action pouvant être définies par l'utilisateur :

- extinction de la chambre des enfants à 9h tous les soirs de semaine ;
- fermeture des volets 30 minutes après le coucher du soleil ;
- allumage de l'arrosage automatique à 6h30 tous les matins de juin à août.

Ces exemples permettent de visualiser la complexité que peuvent avoir les éventuelles règles. Une fois encore, le système de gestion des règles doit donc être le plus générique possible, mais également le plus simple d'utilisation possible.

En plus de cette interface utilisateur, une interface « d'administration » est disponible. Celle-ci a pour but de permettre la gestion de la configuration globale (activer le débogage, la gestion des logs, etc.). L'intégralité de l'interface pourra bien sûr être protégée par une procédure d'accès via mot de passe.

1.3 Installation de Domogik

1.3.1 Interface Web

L'installation du projet Domogik n'est pour le moment pas automatisée. La plus grande partie de l'installation consiste à la mise en place du framework Django afin de faire fonctionner l'interface Web. Pour ceci, une base de données est nécessaire. On commence donc par installer un serveur de gestion de base de données. Nous choisirons ici MySQL :

```
sudo apt-get install mysql-server-5.0
```

Bien sûr, vous devez adapter cette commande à votre distribution. Sous Debian/Ubuntu, vous devrez spécifier le mot de passe pour l'administrateur lors de l'installation. Il est également nécessaire d'installer la bibliothèque Python permettant de communiquer avec MySQL :

```
sudo apt-get install python-mysqldb
```

Il est bien sûr nécessaire d'avoir un interpréteur Python d'installé. Cela est le cas par défaut sur la plupart des distributions. Nous allons maintenant installer le framework Django sur le système. Pour cela, téléchargez la dernière version de ce framework sur le site http://www.djangoproject.com/download/ (version 1.0.2 à l'heure actuelle). Il est ensuite nécessaire de décompresser l'archive :

```
tar xvzf Django-1.0.2-final.tar.gz
cd Django-1.0.2-final
```

Nous allons ensuite installer Django :

```
sudo python setup.py install
```

Il est maintenant nécessaire de créer une base MySQL et un utilisateur correspondant à l'aide de la commande :

```
mysql -u root -p<Mot de passe>
```

Il est bien sûr nécessaire de saisir le mot de passe défini lors de l'installation du serveur MySQL. Vous obtenez alors le prompt de MySQL :

```
mysql>
```

Entrez-y les commandes suivantes

```
create database domogik;
```

qui créeront la base, suivi de

```
create user 'domogik'@'localhost' identified by 'domo2008';
grant all on domogik.* to domogik@localhost;
```

qui créera l'utilisateur **'domogik'** avec le mot de passe **'domo2008'** et lui donnera les droits sur la base.

Vous pouvez ensuite quitter le shell MySQL avec la commande :

```
quit
```

Nous voilà prêts à installer Domogik. Rendez-vous sur le site **http://labs.libre-entreprise.org/scm/?group_id=135** et téléchargez le *snapshot* des sources. Il faut ensuite extraire l'archive téléchargée :

```
tar xvzf domogik-scm-latest.tar.gz
cd domogik-scm-2009-XX-YY
```

Ceci va vous télécharger tout le contenu du svn. Ce qui nous intéresse ici est le sous-répertoire domogik. Éditez le fichier **settings.py** et modifiez les paramètres de connexion à la base si nécessaire.

Si vous avez suivi les lignes précédentes, vous ne devriez pas avoir besoin de changer quoi que ce soit.

```
cd domogik
```

Vous devez commencer par créer la structure initiale de la base de données :

```
python manage.py syncdb
```

Après avoir créé les bases, l'installateur va vous proposer de créer un administrateur Django. Répondez « yes » et répondez aux questions qui vous sont posées. L'installateur va ensuite peupler la base avec les structures du projet Domogik.

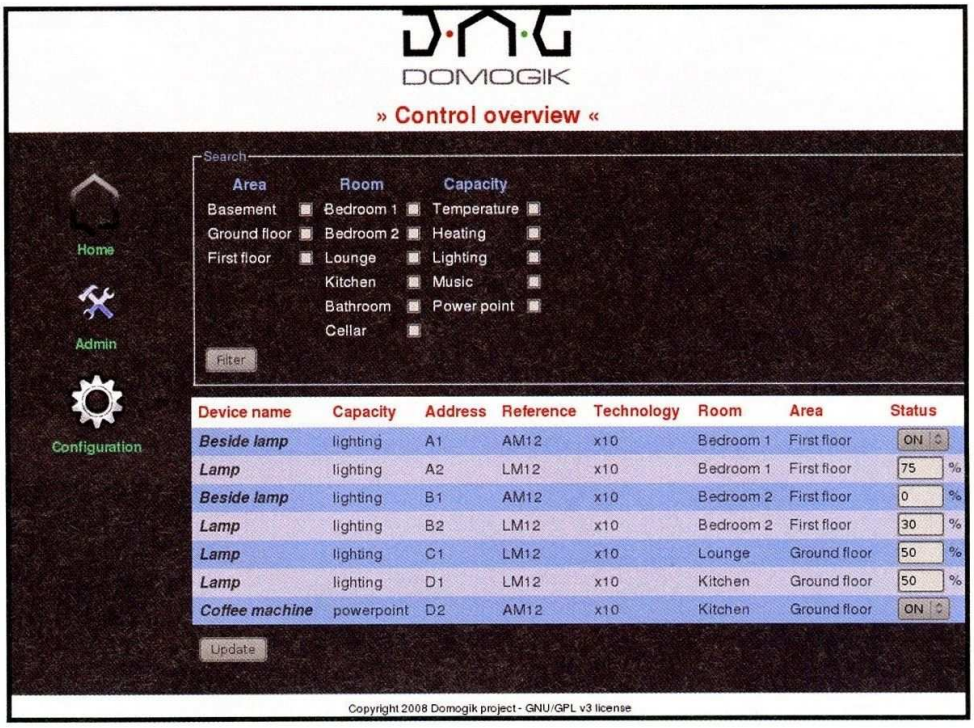
Voilà, Django est maintenant installé. Il ne reste plus qu'à le lancer via la commande :

```
python manage.py runserver
```

Ceci démarrera le *daemon* Django et chargera l'application Domogik. Vous allez maintenant pouvoir accéder à l'interface Domogik (qui a dit « enfin » ?) à l'adresse http://localhost:8000/domogik/. Lors du premier chargement, un message « *No device found !* » apparaîtra. Cela est dû au fait qu'aucun élément n'est défini. Le menu ***Admin*** vous permettra d'utiliser les fonctionnalités avancées de l'interface. Il vous permettra également de charger des éléments factices. Par la suite, si vous désirez ajouter des modules, cela pourra se faire par le menu ***Configuration***.

Rendez-vous donc dans le menu ***Admin*** et cliquez sur ***Load*** dans la section ***Simulation***. Retournez alors dans la section ***Home***. Vous pourrez maintenant y voir des éléments factices. Comme vous pouvez le voir, vous pouvez filtrer vos éléments par type, pièce ou zone.

Il ne vous reste plus qu'à vous amuser avec l'interface.



1.3.2 Réseau xPL

À l'heure actuelle, il n'y a pas d'installateur automatisant le réseau xPL. Il y a déjà eu un article expliquant en détail le protocole xPL dans ce même magazine, ainsi qu'une petite explication ci-dessus. Seule l'installation sera donc détaillée dans les lignes qui vont suivre.

Il est donc nécessaire de démarrer un hub qui sera le point central du réseau xPL. Ensuite, il suffira de démarrer les scripts souhaités.

Le hub le plus simple à utiliser sous Linux est disponible sur le projet xPL4Linux. L'installation se fait de la façon suivante :

```
wget http://www.xpl4java.org/xPL4Linux/downloads/xPLHub.tgz
tar xvzf xPLHub.tgz
```

Enfin, pour lancer le daemon, il vous suffit d'utiliser la commande suivante :

```
./xPL_Hub -debug -xpldebug -nodaemon
```

Ici, nous lançons le hub de façon à pouvoir déboguer le fonctionnement. Plusieurs lignes devraient s'afficher, avec à la fin :

```
INFO: xPL Hub now running
```

Indiquant que tout est OK. Une fois que tout fonctionne, il vous suffira de lancer le programme **xPL_Hub** sans paramètres pour le lancer en tâche de fond.

Maintenant que votre hub fonctionne, il ne vous reste plus qu'à démarrer les *plugins* souhaités. Pour cela, rendez-vous dans le répertoire **xpl** contenu dans l'archive du projet Domogik. Vous y trouverez quelques plugins, qu'il suffit de lancer comme un simple binaire. Suivant votre configuration réseau, il peut être nécessaire d'éditer chaque script et de changer le paramètre **ip** dans l'appel au constructeur « Manager ».

Le script **main_DawnDusk.py** ne nécessite aucun matériel domotique. Vous pouvez donc le lancer après avoir éventuellement modifié l'adresse IP via la commande :

```
./main_DawnDusk.py
```

Dès le lancement, vous devriez voir de nombreuses lignes apparaître dans la console dans laquelle vous avez lancé le hub xPL, indiquant la connexion du plugin et l'émission d'un message.

Chaque plugin fonctionnera de la même façon.

1.4 Évolution et participation

Le projet Domogik est jeune. Lors de la création, 2 personnes y participaient. Depuis, plusieurs autres contributeurs sont venus rejoindre le projet. À l'heure actuelle, la technologie majoritairement utilisée est le Python. La partie *backend* n'utilise que du Python pour le développement des modules. L'interface web étant développée à l'aide du framework Django, le langage Python est également utilisé. Cependant, comme dans toute interface web, les langages XHTML, CSS et Javascript sont également présents.

Le projet étant en pleine évolution, toute personne souhaitant y participer et connaissant (ou souhaitant apprendre) un des langages ci-dessus est libre de rejoindre le projet. La partie graphique étant également importante, le projet est aussi à la recherche de personnes possédant des compétences dans ce domaine.

À l'heure actuelle, de nombreuses évolutions importantes sont à prévoir prochainement. On peut citer l'utilisation de fichiers de configuration pour les plugins, l'automatisation des installations, la mise en place de simulateurs de matériels, et, bien sûr, la mise en place de la liaison entre l'interface Web et les plugins.

Bien entendu, toute personne souhaitant participer à ces tâches est la bienvenue.

Auteur : Maxence Dunnewind

Fiche d'identité

- Projet : Domogik
- Site web : **http://www.domogik.org**
- Plateforme de développement : **http://labs.libre-entreprise.org/projects/domogik/**
- Liste de diffusions : **http://lists.labs.libre-entreprise.org/mailman/listinfo/** (domogik-general, domogik-developers et domogik-commits)
- IRC : **#domogik sur irc.freenode.net**

Liens

- Framwork Django : **http://www.djangoproject.com**
- Communauté française de Django : **http://www.django-fr.org**
- Projet xPL : **http://wiki.xplproject.org.uk**
- Projet xPL4Linux : **http://www.xpl4java.org/xPL4Linux**

ENVIE DE PUBLIER FACILEMENT SUR LE WEB ?

LINUX PRATIQUE HS 17

AU SOMMAIRE...

ENCORE DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 17 AVRIL 2009

Parce qu'y'en a marre !

Y'en a marre des discours marketing qui incitent les développeurs à faire n'importe quoi ! Et ils s'y précipitent sans recul, avec la certitude d'avoir fait les bons choix, alors qu'ils ne font que suivre des discours lénifiants. Où sont les petits artisans qui façonnent un code avec de bons produits et qui réfléchissent avant d'agir ? Coup de gueule de Jean-Pierre Troll...

Auteur

- Jean-Pierre Troll

1 L'excès de structures

Y'en a marre des grands principes d'architectures qui imposent des couches et couches sans valeurs ajoutées. Où est perdu le sens commun ?

« Il faut séparer la couche métier, de la couche processus, de la couche présentation, de la couche de persistance. Chaque couche doit être hermétique. »

Super. J'ai participé à des projets basés sur ce type d'approche.

- Comment gérer les exceptions ?
- Eh bien, dans chaque couche, on capture l'exception, on l'encapsule dans une autre et on la propage.
- On ne doit pas capturer les exceptions par leurs types ? Si je l'encapsule, je n'ai plus le moyen de différencier la gestion d'erreur suivant son type !
- Si, si. Fais un *switch* sur le `instanceof getCause()` !
- Ce n'est pas le rôle de l'instruction `catch` ?
- Euh, oui, mais autrement cela viole les principes architecturaux.

Donc, pour écrire un simple traitement métier pouvant par mégarde avoir un problème (violation d'une règle d'intégrité de la base de données, paramètres invalides, etc.), je dois écrire beaucoup de traitements, sources d'erreurs.

J'écris le traitement de la couche métier qui traite les paramètres et retourne un objet métier. Un nouveau traitement, normal.

Dans la couche métier, une capture d'exception de la couche de persistance. Transformer l'exception et la propager. Et un nouveau traitement, cela fait deux.

Pour qu'elle ait une sémantique, je dois donc écrire une nouvelle classe d'exception spécifique. Et une nouvelle classe!

Dans la couche processus, s'il n'y a pas d'exception, je vais bien entendu transformer les objets de présentation reçus en paramètres pour les rendre présentables à la couche métier. Je fais de même pour transformer les objets de retour de la couche métier en objets utiles à la couche présentation. Je remplace les entiers, les flottants, etc. en chaînes de caractères. Et deux nouvelles classes, ce qui fait trois et deux nouveaux traitements, ce qui fait quatre.

De plus, je capture l'exception pour l'encapsuler et la propager sous une autre forme. Et un nouveau traitement, ce qui donne cinq.

Pour garder une sémantique, je dois écrire une nouvelle classe d'exception spécifique à la couche processus. Et une nouvelle classe, ce qui fait quatre.

Dans la couche présentation, j'utilise les objets de présentation. En cas d'exception, j'analyse en profondeur la cause, puis je présente des excuses à l'utilisateur. Un traitement de plus, mais c'est inévitable.

Pour respecter l'architecture, j'ai donc écris cinq traitements supplémentaires et quatre classes. Tout cela pour une seule petite méthode métier !

Les impacts négatifs sont nombreux : complexification du code ; multiplication du nombre de lignes de code, donc d'erreurs ; difficulté à faire évoluer le code ; empreinte mémoire plus importante ; consommation CPU plus importante ; multiplication des ressources de la plateforme de production ; impacts écologique associés.

Rappelez-moi l'objectif d'une architecture ? N'est-ce pas d'uniformiser et de simplifier ? C'est réussi ? Il semble que non.

Pourquoi ne pas utiliser des idées simples comme : « Les objets métier peuvent être directement manipulés par la présentation ». « Les objets métier sont persistants » « Les exceptions ne doivent en aucun cas être

converties, à moins d'apporter un valeur ajoutée permettant un meilleur traitement de celle-ci. »

Certaines idées émergent maintenant, comme le mariage de la couche métier et de la couche de persistance, grâce à des *frameworks* d'annotation et de modification dynamique du code. C'est une bonne chose. Mais, sans ces frameworks, c'était possible également. Par excès de structures, on a séparé les couches.

Lorsque l'on propose des solutions alternatives à ces architectures, démontrant clairement la simplification d'une architecture alternative, la réduction du nombre de lignes de code, l'amélioration des performances, on reçoit généralement un refus du client ou de l'architecte. « C'est moins académique... » Que répondre ? Il n'y a pas plus sourd que celui qui ne veut pas entendre.

Nous manquons de bon sens ou de sens pratique. Les études que nous avons suivies nous incitent tous à faire la même chose, sans réfléchir. Comme le montre brillamment l'expérience de Victoria Horner (http://tinyurl.com/8n7wx2), les singes réfléchissent, les enfants nous singent. Il est temps de devenir adulte et de réfléchir.

À suivre le troupeau, on fonce vers des catastrophes. Dans un autre métier, la finance par exemple, il fallait surtout suivre le voisin, même si tout le monde savait qu'on allait dans le mur. Maintenant, c'est fait.

On ne vous reprochera pas d'avoir tort avec les autres. Seulement d'avoir tort tout seul. Par contre, avoir raison tout seul n'est pas facile.

Autre exemple réel. Je devais faire passer des tests en C++, rédigés par d'autres, afin de tester les connaissances des candidats à l'embauche. La question était simple : écrivez une classe qui permet d'avoir une pile de caractères avec les méthodes `push()` et `pop()`. (Les *Standards Templates Librairies* n'existaient pas encore). Une méthode pour ajouter un caractère à la pile, une autre pour enlever le caractère au-dessus de la pile.

Tous les candidats sans exception écrivaient une classe périphérique `Item` pour pouvoir construire une liste chaînée portant le caractère à mémoriser. La classe `Stack` offrait les méthodes `push()` et `pop()` et manipulait la liste chaînée en conséquence.

Super, cela fonctionne. Mais, c'est la plus mauvaise réponse pour le cas d'espèce. Il s'agit d'une pile de caractères ! Soit un seul octet en mémoire ! Mettre en place une infrastructure logicielle lourde, avec de nombreux objets, des pointeurs partout est une absurdité. C'est utiliser trois semi-remorques pour aller chercher une demi-baguette.

La bonne réponse est d'allouer un simple tableau de caractères et de gérer un entier indiquant le nombre d'éléments. Si le tableau déborde, on en crée un nouveau plus grand, on recopie l'ancien, on ajuste les compteurs et voilà.

La réponse académique n'est pas la bonne. Pour s'en rendre compte, il faut réfléchir un peu et ne pas foncer tête baissée. A-t-on appris à réfléchir dans nos formations ? Allons-nous continuer comme des enfants ou avons-nous passé le stade du singe ?

L'inversion de dépendance est la dernière idée à la mode. C'est en effet une excellente solution pour casser la dépendance entre les composants, pour pouvoir les réutiliser dans différents contextes. C'est très bien pour les composants génériques et les *frameworks*. Mais est-ce vraiment utile pour les simples applications ?

Pour une application, cela permet de lier quelques objets entre eux, le temps d'un test unitaire. Les tests sont ainsi plus légers. Mais, en production, les objets à utiliser et à lier sont finis, bien identifiés et ne doivent pas bouger. On ne change pas à chaud de framework de persistance ! Utiliser l'inversion de dépendance dans la phase d'initialisation de l'application pour construire les différents objets et les lier entre eux est une bonne idée. Cela prend quelques lignes de code.

Faire la même chose via un fichier XML complexe, entraînant l'utilisation de framework lourd, cela en vaut-il vraiment le coup ? J'ai toujours pensé qu'il était plus facile d'écrire quelques lignes de code Java ou C# que d'écrire la même chose en XML. Le rapport signal/bruit du XML étant très, très mauvais, quelques lignes de Java ou C# sont toujours plus économiques en volume de code, vitesse d'exécution et place mémoire.

Par abus du principe d'inversion de dépendance, les projets ne savent plus lier deux objets sans passer par le framework Spring. Il faut que tous les objets puissent être reliés à tous les autres objets du monde. Est-ce vraiment utile ? Faut-il vraiment que tous les objets utilisent Spring ? Les quelques objets structurant ne sont-ils pas suffisants ? Faut-il un fichier XML énorme pour montrer qu'on maîtrise le framework ?

Maintenant, l'évolution consiste à utiliser les annotations pour cela. C'est une très bonne chose. Il faut abandonner les fichiers de paramètres XML si on peut écrire la même chose avec quelques lignes de code !

Lorsque l'on conçoit une architecture logicielle, il faut toujours garder à l'esprit le volume de code nécessaire à écrire pour la respecter. Une bonne architecture est économe en ressources. Elle doit être pragmatique et non académique.

La différence entre un terroriste et un architecte ? On peut négocier avec un terroriste.

À quand une charte de qualité environnement pour les projets informatiques ? Il faut exiger une consommation de ressource minimum dans les appels d'offres. Valoriser les améliorations du code permettant de libérer un des serveurs du parc informatique et contribuer à sauver la planète.

Parce qu'y'en a marre de la médiocrité, des développeurs en batterie et de la disparition des petits artisans soucieux du travail bien fait.

Auteur : Jean-Pierre Troll

Faites vos jeux !

Auteur

- p-e-g

Papi regarde comment qu'on arrive à tuer le méchant 3ème niveau dans « Total violence annihilate the world of the death ». T'as vu comment il est vraiment bien ce « quake like », les mesh cell shadés sont superbes, et on a jamais fait mieux. Oh là les enfants, avant de vous ébahir sur ce truc, laissez-moi vous parler de l'époque des super héros du « code ». Cette époque où les hackers arrivaient à faire prendre souche à leurs rêves avec des machines dites « MicroLoisir ». C'était la bonne vieille époque des space invaders et des « Pacman ». Bref, papi pourrait vous parler du bonheur du clavier du TO7/70 ou encore de l'assembleur de l'Atari 520 ST, mais, surtout, les enfants, n'oubliez pas que le monde du jeu vidéo a débuté il y a bien bien longtemps.

Dans notre nouvel appartement qui est vachement plus grand que l'ancien, on a décidé de regarder un film avec ma copine. Comme pas mal de couples, on est un poil suréquipé en trucs média : la TV, le PC qui sert à écouter la radio du web, la mini-chaîne hifi, la box ADSL, le décodeur ADSL vers TV et surtout un lecteur de DVD. Hier soir, on avait décidé de regarder un film du PC sur la télé. En théorie, la situation était relativement simple, il suffisait de brancher la sortie du PC sur l'entrée numérique de la TV et de lancer un lecteur multimédia. En fait, la situation est moins évidente dans la mesure où notre TV ne possède pas d'entrée numérique pour lire un film. Bref, la soirée film sur PC était compromise et la plupart des couples seraient allés chercher un DVD à mettre dans le lecteur. Ça, c'est la situation du couple dit « normal », qui ne possède pas un vrai codeur chez soi.

Chez nous, la situation a mal tourné. Mon œil d'informaticien aguerri remarque que la box ADSL est branchée sur le décodeur ADSL vers TV et que le PC est branché sur la même boîte. Je me dis donc que, simplement, il va me suffire de brancher directement le PC au décodeur et d'envoyer dessus mon film, comme s'il s'agissait d'une chaîne provenant de l'ADSL. Jusque-là, la situation est pas mal maîtrisée et j'explique relativement fier de moi à mon amie que ni une, ni deux, quelques câbles, un peu de hacking et ni vu, ni connu, on regarde le film du PC sur la TV dans 5/10 minutes.

Je mets le plan sauvage en pratique, et vais récupérer un câble Ethernet de 10 m que je branche entre le PC et la boîte, et je simule avec mon PC un décodeur ADSL. Le but étant de sniffer avec Ethereal les ports UDP utilisés par la TV sur Internet. Je remarque une IP et un port UDP et hop je rebranche le décodeur avec le PC. Inutile de dire qu'à ce moment-là il y a au milieu du salon un PC, un décodeur ADSL et une box qui traînent au milieu des 10 m de câble Ethernet. L'un dans l'autre, mon amie commence à ouvrir un magazine.

Je prends mon VLC et commence à envoyer le film en UDP sur le port utilisé par la chaîne Internet. Après pas mal de temps et de réflexion, je commence à avoir le son du film sur la TV, mais pas l'image. Je me dis que c'est peut-être normal, j'ai utilisé une vieille version de VLC sous Windows. J'annonce à mon amie que je vais chercher un portable sous Unix avec une connexion Wifi pour récupérer et recompiler VLC et ça marchera ensuite. Elle, de son côté commence à prendre une collection de BD, car elle sait que cela ne sera peut-être pas aussi rapide que prévu.

Après recompilation du VLC, lancement du film, toujours pas de résultat et je commence à me demander ce qui peut bien se passer ? Serait-ce encore un codec foireux ou un pilote quelconque qui ne marcherait pas ? Je me retourne pour chercher au fond des yeux de ma copine une réponse, et celle-ci me tend un DVD à mettre dans le lecteur. L'un dans l'autre, les 5/10 minutes de branchement ont duré presque 2 heures et bien que je considère le DVD comme étant une technologie du siècle passé, je cède à cette pression. Mais soyons clairs, pour la prochaine soirée film sur PC, je coderai un vague truc en SDL pour décoder le film et l'envoyer sur le décodeur...

1 SDL ou le retour du vieux codeur

L'ancien temps fut une période bénie pour la programmation des jeux vidéo. Il y avait une carte graphique VGA, un système d'exploitation grand public (MS DOS) et une ou deux manières d'adresser le matériel. Cette simplicité architecturale permettait d'accéder directement au matériel afin d'afficher des images, gérer les interactions utilisateur et enfin fabriquer un jeu vidéo. L'accès direct au matériel permettait de mettre en œuvre deux opérations spécifiques au jeu vidéo 2D, le « *blit* » et l'accès direct au *framebuffer*. Le blit est une opération consistant à copier un morceau ou la totalité de l'image dans une autre image. Cette opération basique permet par exemple de copier un personnage animé sur un fond d'écran. Si on prend l'exemple d'un petit personnage dans un jeu, lorsque l'utilisateur demande à aller à droite, on « blitte » un fond d'écran, puis on blitte le personnage. L'autre opération nécessaire pour faire un jeu vidéo est un accès direct au framebuffer ou, autrement dit, à la surface que l'on souhaite afficher. Cette nécessité vient du fait que, pour faire un jeu vidéo, il faut que les opérations effectuées pour l'affichage soient très rapides. Cette rapidité permet d'économiser du temps pour faire des interactions utilisateur rapides ou encore jouer de la musique ou bien faire quelques effets spéciaux.

Bref, c'était une époque bénie, une carte, un *buffer*, un OS et une quantité d'astuces pour effectuer des blits rapides et une quantité d'astuces pour fabriquer le plus beau jeu vidéo du monde. La situation a pas mal évolué : le nombre et les capacités des cartes graphiques ont explosé, les OS se sont démultipliés (Windows, Linux, BSD Like, Solaris... pour ne citer que les plus connus), et ceci a rendu pendant longtemps la programmation de jeux vidéo plus difficile. La question n'est plus de savoir comment faire un blit rapide, mais d'arriver à le faire de façon à ce que le jeu soit portable sur toutes les cartes et les OS. De façon plus cruelle, arriver à avoir accès au matériel de façon directe est quasiment impossible car :

- On ne sait pas à quel type de carte on cause (nVidia, ATI...?)
- L'OS protège le matériel d'un accès direct par une application.

On pourrait se dire que la puissance des machines actuelles permet de faire un jeu vidéo de façon générique sans avoir à utiliser le framebuffer de façon trop directe. Après tout, il y a plein de langages comme Java qui permettent de jouer avec des images de façon portable. Malheureusement, la couche d'abstraction apportée par ces technologies est trop importante pour permettre d'avoir des animations complexes et fluides ce qui est nécessaire pour faire un jeu vidéo.

Heureusement, depuis quelques années, une bibliothèque **SDL** permet d'accéder de façon relativement directe et simple au matériel. Celle-ci autorise la réutilisation des techniques d'antan afin de fabriquer un jeu.

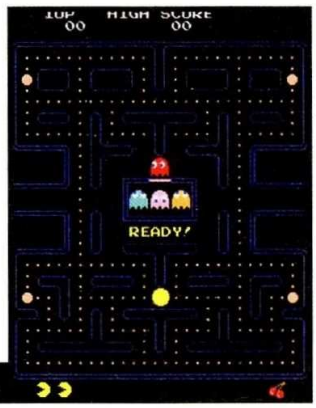
Pour cette série d'articles, le lecteur aura la bonté de bien vouloir télécharger et installer les bibliothèques **SDL** et **SDL_image** se trouvant respectivement sur http://www.libsdl.org/ et http://www.libsdl.org/projects/SDL_image. Que ce soit sous Win32 ou sous Linux, j'ai pris les versions binaires de ces bibliothèques et utilisé le compilateur GNU g++.

Le but de cette série d'articles n'est pas de découvrir une fois de plus cette bibliothèque déjà bien connue, mais de voir comment faire un jeu à travers elle.

2 Pacman Enflammé ou « Fire Rage of the Pacman of the death »

Le but de cette série d'articles est de voir comment on peut utiliser SDL pour faire un vrai mini-jeu inspiré de Pacman. En effet, sur l'Internet, on trouve pas mal de tutoriels sur « comment faire *hello world* » en SDL, mais aucun ne montre comment se fabrique de bout en bout un jeu. C'est ici qu'intervient cette série d'articles qui blocs après blocs va tenter avec plus ou moins de bonheur de démystifier les techniques du jeu vidéo 2D.

Dans sa version classique, un personnage « Pacman » est enfermé dans un labyrinthe, mange ses pastilles et se fait courser par des fantômes.



Source : http://en.wikipedia.org/wiki/File:Pac-man.png

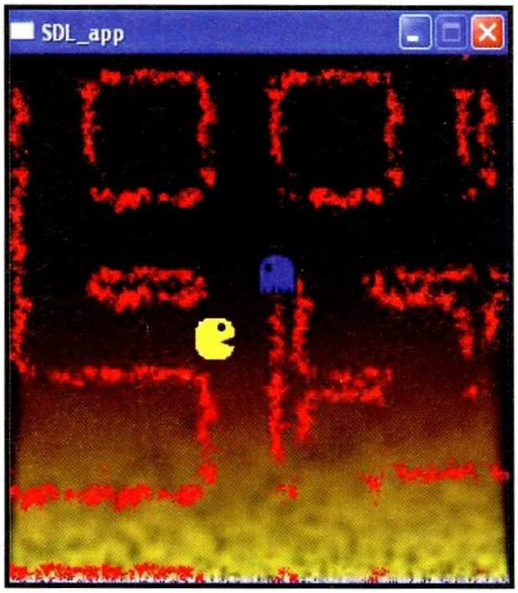
D'un point de vue technique, pour arriver à ce résultat, il faut pouvoir :

- Gérer l'affichage et l'animation du Pacman (dont la bouche passe son temps à s'ouvrir et à se fermer) ainsi que des fantômes.
- Une fois que l'on a réussi à l'afficher, il faut pouvoir gérer les « collisions » avec les murs du labyrinthe et les pastilles.
- Essayer de donner un comportement « intelligent » aux fantômes.

En partant de là, nous avons un début de jeu classique, mais afin de ne pas se limiter à un jeu de plus 30 ans, nous allons en faire une interprétation très libre et rajouter pas mal de fonctionnalités « nouvelles » :

- Les murs du labyrinthe ne sont pas propres et droits, mais déchiquetés et tordus. Ceci change pas mal de choses sur la détection des collisions, car, autant il est simple de tester une collision avec un mur droit, autant c'est un poil plus complexe de faire ça avec un mur tordu.
- Le labyrinthe n'est pas visible en totalité sur l'écran. Dans notre jeu, le labyrinthe est beaucoup plus grand que la fenêtre de jeu, et donc ladite fenêtre bouge en même temps que le Pacman. En termes techniques, il s'agit d'un *scrolling* multidirectionnel.
- Enfin, un Pacman ne marche pas sur un fond de labyrinthe tout noir, mais sur un mur de flammes qui évolue en permanence. Il s'agit d'un des effets spéciaux les plus faciles à faire.

Nous profitons de cette description pour fournir une capture d'écran de notre Pacman :



Donc, notre Pacman marche sur un fond en flamme, et c'est cette fonctionnalité qui donne son nom a notre jeu : *Pacman Enflammé* (ou P.E. dans la suite de l'article).

Note de l'auteur à un éventuel éditeur de logiciel : Le résultat est vachement joli, et qui sait, ptêt ben que je vais réussir à le vendre mon *Pacman Enflammé*. Comme ce n'est pas le nom le plus vendeur du monde, il faudra peut être le *repackager* avec un nouveau titre style « *Fire Rage of the Pacman of the death* » :).

Blague et bêtise à part, le but de ce premier article n'est pas de tout montrer, car cela rendrait l'article indigeste, mais d'apprendre à utiliser SDL afin de bouger un simple Pacman à l'écran. Dans des articles à venir, nous rajouterons les briques nécessaires pour aboutir à ce résultat.

3 Principale fonction de la bibliothèque SDL

Le but de cette partie n'est pas de voir la totalité de la bibliothèque SDL, mais d'en voir suffisamment pour pouvoir mettre en œuvre notre P.E. La première fonction à appeler pour que SDL puisse commencer à travailler est la fonction **SDL_Init** qui prend un jeu de drapeaux en paramètre. Cette fonction permet d'initialiser un ou plusieurs modules de la bibliothèque en fonction de la valeur passée en argument. Le paramètre **SDL_INIT_VIDEO** demande à initialiser le module graphique, **SDL_INIT_AUDIO** le module de son et **SDL_INIT_TIMER** la gestion du temps. La documentation des API de SDL donne la totalité des paramètres possibles.

Une fois cette initialisation faite, il faut, si on souhaite dessiner à l'écran, demander à SDL de nous donner une surface de dessin.

La fonction **SDL_SetVideoMode** permet d'obtenir une surface sur l'écran pour faire un joli dessin. Cette fonction prend en paramètre la longueur et la largeur de la surface désirée, ainsi que le nombre d'octets par pixel que l'on souhaite. Ce dernier paramètre a déjà été vu dans la série sur les compilateurs LOGO. Nous n'en donnerons qu'un rapide aperçu ici. Un pixel coloré peut se décomposer en plusieurs couleurs de base, et une décomposition relativement connue est la norme RGB (pour *Red, Green, Blue*). Donc, pour coder un pixel coloré, il faut un codage sur plusieurs octets pour coder la partie rouge, verte et bleue dudit pixel. Actuellement, on prend un octet pour chaque composante, ce qui donne 24 bits pour un pixel. Une autre façon de coder est de prendre la décomposition du pixel en 4 composantes, 3 composantes RGB et une composante de transparence. Le nombre de bits par pixel est dans ce cas de 32.

Enfin, le dernier paramètre est le type de surface que l'on souhaite avoir. On peut souhaiter que celle-ci soit construite en utilisant de la mémoire centrale de la machine (paramètre **SDL_SWSURFACE**) ou encore dans la mémoire de la carte vidéo (paramètre **SDL_HWSURFACE**). Chacun des deux types présente des avantages et des inconvénients en termes de facilité d'accès aux pixels ou encore en rapidité d'affichage. Enfin, il est possible de cumuler ce dernier paramètre avec le drapeau **SDL_DOUBLEBUF** qui permet de faire du « double buffering » sur la surface. Sans entrer dans les détails, une surface « double buffer » utilise deux fois plus de mémoire qu'une surface « normal », mais permet des animations plus fluides.

Le code de notre P.E. commence donc par :

```
//Nous initialisons la bibliothèque avec les sous modules vidéo, audio
et de gestion du temps
SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO);
// et nous créons une surface de 320x320 pixel avec 4 octets par pixel
// La surface est créée dans la mémoire de la carte vidéo avec un
double buffer pour une
// animation fluide
SDL_Surface *    screen = SDL_SetVideoMode( 320,320,32, SDL_HWSURFACE
|SDL_DOUBLEBUF );
```

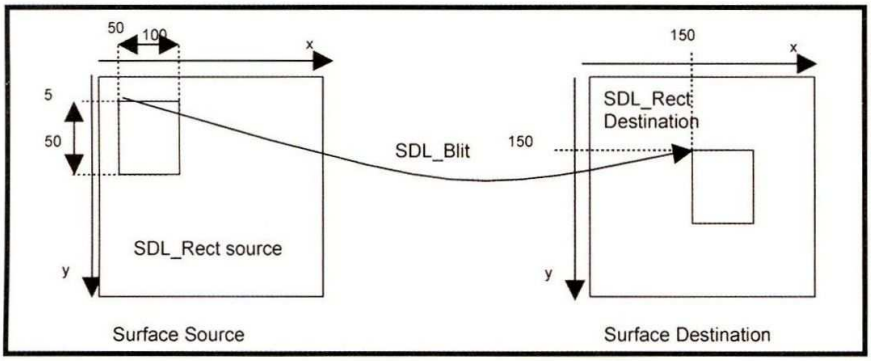
Une autre fonction super utile pour notre Pacman est celle permettant de créer une surface pour travailler nos dessins, la fonction **SDL_createRGBSurface** qui prend en paramètre tout comme la fonction **SDL_SetVideoMode**, la longueur et la largeur de la surface, la taille en octets d'un pixel ainsi que les options de création de surface. Cette fonction prend également en paramètre la manière dont SDL doit interpréter les octets de chaque pixel. Supposons qu'un pixel soit codé sur 32 bits (4 octets). Il faut passer à la bibliothèque les différents masques binaires permettant à SDL d'extraire d'un pixel les composantes rouge, verte et bleue. Dans le cas de P.E., nous créons toutes nos surfaces avec 32 bits par pixel et donc 1 octet par composante R,G,B et alpha. Il faut juste faire attention au fait que la machine qui exécute notre code soit « *big* » ou « little endian ».

Donnons un exemple de code permettant d'y voir plus clair :

```
//Si un pixel est codé sur 4 octets, et que la machine est big endian
alors le premier octet est le
//rouge, le deuxième le vert et le troisième et quatrième sont le bleu
et la composante de
//transparence
// Dans le cas ou la machine est little endian, les composantes sont
inversées
#if SDL_BYTEORDER == SDL_BIG_ENDIAN
   unsigned int rmask = 0xff000000;
   unsigned int  gmask = 0x00ff0000;
   unsigned int bmask = 0x0000ff00;
   unsigned int amask = 0x000000ff;
#else
   unsigned int rmask = 0x000000ff;
   unsigned int  gmask = 0x0000ff00;
   unsigned int bmask = 0x00ff0000;
   unsigned int amask = 0xff000000;
#endif
// On créé une surface dans la mémoire vidéo de 320x320 pixel avec 4
octets de codage
SDL_Surface * masurface=SDL_CreateRGBSurface(SDL_HWSURFACE, 320,320,32,
rmask,  gmask,  bmask,  amask);
```

Il nous reste deux principales fonctions à voir pour commencer à implémenter notre Pacman, la fonction **SDL_BlitSurface** et la fonction **SDL_Flip**.

La fonction **SDL_BlitSurface** est une des fonctions principales que nous allons utiliser, car elle permet de copier des surfaces. Cette fonction prend deux surfaces en paramètre et copie un morceau de la surface source (du morceau donné via la structure **SDL_Rect** que nous allons voir) vers un endroit de la surface destination (ici aussi décrit par un **SDL_Rect**). La structure **SDL_Rect** est juste une structure générique représentant un rectangle avec ses coordonnées X et Y de départ, ainsi que sa largeur et sa hauteur. La compréhension de cette opération étant cruciale pour la suite de cet article, l'auteur se fend ici d'un petit schéma :



À la lecture de ce schéma, le lecteur aura bien compris que l'origine des axes se trouve en haut à gauche de la surface. De plus, les abscisses augmentent de gauche à droite, et les ordonnées sont orientées de haut en bas. Sur une surface de X par Y pixels, le point 0,0 est donc en haut à gauche et le point X,Y en bas à droite.

Reprenons notre schéma en exhibant le code SDL correspondant :

```
SDL_Rect srcRect, dstRect;
srcRect.x=50; srcRect.y=5; srcRect.w=100; srcRect.h=50;
dstRect.x=50;dstRect.y=150;
SDL_BlitSurface ( surfarceSource,&srcRect,surfaceDestination,&dstRect);
```

Il ne nous reste plus qu'à regarder l'opération **SDL_Flip** qui prend une surface et qui l'affiche à l'écran. Le prototype de cette méthode est simplissime : **SDL_Flip(SDL_Surface *)**.

Nous donnons ici un exemple complet et commenté invoquant toutes nos primitives. Cet exemple bien que simple doit afficher une image à l'écran et constitue le squelette de notre Pacman.

```
int main(int argc, char *argv[])
{
   //Initialisation de SDL
  SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO);
  //Creation de l'écran en 320x320 pixel de 4 octet
  SDL_Surface * screen = SDL_SetVideoMode( 320,320,32, SDL_HWSURFACE
|SDL_DOUBLEBUF );
  //On utilise une primitive SDL pour charger une surface
  // contenant l'image bitmap de mybackground.bmp
  SDL_Surface * background=SDL_LoadBMP("mybackground.bmp");
  while (1)
  {
    // la fonction SDL_PollEvent renvoie vrai si un événement
    // a été déclenché (mouvement de souris, touche de clavier..)
    while (SDL_PollEvent( &event ))
    {
    //Si on est ici c'est que nous avons eu un événement
    switch (event.type)
    {
      // Si le type est une touche pressée au clavier
      // on rentre dans ce cas
      case SDL_KEYDOWN:
        //Si cette touche pressée est la lettre q
        // on ferme le programme
        switch( event.key.keysym.sym )
        {
          case 'q': exit(0); break;
          default : break;
        }
      }
    }
    // que l'on ait eu un événement ou pas, on copie
    // la surface contenant l'image dans la surface
       de l'écran
    SDL_BlitSurface(background,NULL,screen,NULL);
    // et on affiche l'écran
    SDL_Flip( screen );
    }
  }
```

4 Les sprites

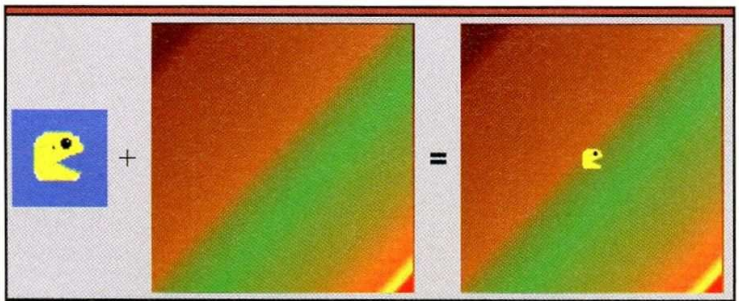
Les *sprites* représentent les objets mouvants et animés d'un jeu. Par exemple, un Pacman et le fantôme de notre jeu sont des sprites. Nous allons voir dans ce chapitre comment coder le sprite du Pacman qui possède deux caractéristiques :

- Si la tête du Pacman est orientée dans la direction du mouvement. autrement dit, si le Pacman va de gauche à droite, alors sa tête est orientée à droite.
- Quoi que notre Pacman fasse, celui-ci passe son temps à ouvrir et à fermer la bouche.

Donc, nous allons jouer avec un sprite (celui du Pacman) pour voir comment ça se passe. La première chose à faire est de présenter la technique de « l'écran bleu » de la météo. Lorsque l'on regarde la météo, le/la présentateur/trice bouge devant une carte satellite, puis une carte de France du temps. Comment ça se fait que l'arrière-plan puisse changer aussi facilement ? C'est la technique du fond d'écran bleu.

En fait, on filme la personne devant un écran tout bleu (la personne qui présente gesticule dans le vide), on capture les images en numérique, puis on remplace numériquement tous les pixels bleus par la bonne carte.

Afficher un sprite, c'est exactement la même chose. Un sprite est dessiné sur une petite surface dont le fond est complètement uni avec une certaine couleur (ici le bleu). Ensuite, on demande à la bibliothèque SDL de coller la surface contenant le sprite dans une autre surface (ici un dégradé de vert et rouge) en lui donnant la couleur de transparence à ignorer. SDL copie donc les pixels de notre sprite sur la surface de fond en omettant la couleur « bleu ».

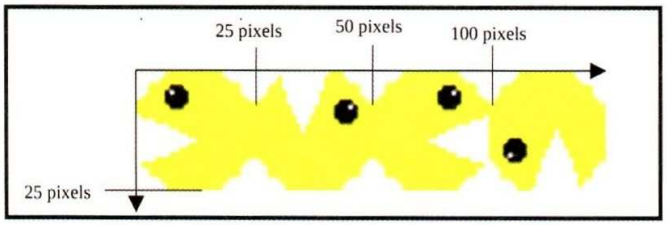


4.1 La version statique du sprite

Maintenant que c'est super bien compris, il faut voir comment on gère le fait que le Pacman ait la tête en haut/bas/gauche/droite lorsque celui-ci va en haut/bas/gauche/droite.

Pour arriver à cela, on se sert d'une planche de dessin qui contient les différents mouvements du Pacman.

Nous donnons ici la planche de sprite :



Sur notre planche, la hauteur des sprites est de 25 pixels et, tous les 25 pixels, nous avons une autre position du Pacman.

Donc, nous chargeons notre image contenant tous les mouvements du Pacman, et lorsque le sprite va vers :

- la gauche, alors nous effectuons un blit de la surface de 0x0 jusqu'à 25x25.
- le haut, notre blit ira de 25x0 jusqu'à 50x25.
- etc. pour les autres mouvements.

Nous définissons donc une classe de sprite qui a besoin de se souvenir de 2 états :

- Les coordonnées du sprite sur la surface de fond d'écran.
- Son mouvement afin de pouvoir blitter correctement une sous-partie de la surface de notre planche de sprite à l'écran.

Nous allons maintenant nous attaquer à l'implémentation de ce sprite. Le but ici est d'avoir une implémentation simple plutôt que performante afin d'avoir une bonne compréhension de ce qui se passe. Nous avons choisi le langage d'implémentation C++ dont l'approche objet est bien adaptée.

Donc, voici l'interface commentée de la classe **Sprite** :

```
#ifndef __SPRITE_H
#define __SPRITE_H
#include <SDL/SDL.h>
#include <string>
class Sprite
{
   private:
   // position du sprite a l'écran en X et en Y
   unsigned int xpos;
   unsigned int ypos;
   // la direction du sprite
   unsigned int direction;
   // le nombre de sprites sur la planche
   unsigned int nbSpriteX;
   // la taille en pixels d'un sprite sur la planche
   unsigned int spriteWidth; unsigned int spriteHeight;
   // la planche de sprites au format SDL_Surface
   SDL_Surface * spriteMap;
   public:
   // Constructeur prenant en paramètre le nom du fichier
   // image contenant les sprites
   // le nombre de sprites sur la planche en X
   // la taille en pixels d'un sprite sur la planche
   Sprite(std::string & filename,  unsigned int nbSpriteX,
unsigned int spriteWidth,
   unsigned int spriteHeight);
   // constructeur par copie
   Sprite(const Sprite & sprite);
   //Dessine le sprite sur la surface "surface"
   // à l'emplacement donné par xpos,ypos
   // avec l'image issue de spriteMap
   void drawMe(SDL_Surface * surface);
   //Accesseur
   unsigned int getSpriteWidth(); unsigned int getSpriteHeight();
   unsigned int getDirection();
   void changeDirection(unsigned int direction);
   void setPos(unsigned int x, unsigned int y);
   unsigned int getX();  unsigned int getY();
   // destructeur
   ~Sprite();
};
#endif
```

Cette interface **Sprite.h** est relativement claire et nous allons donner l'implémentation des deux méthodes un poil complexes de cette classe. En premier lieu, nous donnons le constructeur de cette classe :

```
#include "Sprite.h"
#include "SDL_image.h"
Sprite::Sprite(std::string & filename, unsigned int nbSpriteX, unsigned
int spriteWidth,unsigned int spriteHeight)
{
   SDL_Surface* loadedImage=IMG_Load(filename.c_str());
   this->spriteMap  = SDL_DisplayFormat( loadedImage );
   SDL_FreeSurface( loadedImage );
   Uint32 colorkey = SDL_MapRGB( spriteMap->format, 0xFF, 0xFF, 0xFF );
   SDL_SetColorKey( spriteMap, SDL_RLEACCEL | SDL_SRCCOLORKEY, colorkey
);
   this->nbSpriteX=nbSpriteX;
   this->spriteWidth=spriteWidth;this->spriteHeight=spriteHeight;
   this->direction=0;this->xpos=0;this->ypos=0;
}
```

Nous allons détailler les lignes critiques de cette méthode. En premier lieu, on remarquera l'inclusion du fichier **SDL_Image.h**. Ce fichier vient d'une bibliothèque additionnelle à SDL (qui s'appelle **SDL_Image**) et qui ne fournit qu'une seule fonction **IMG_Load(const char *)**. Cette fonction prend un nom de fichier contenant une image (au format BMP, GIF, JPEG et PNG) et renvoie la surface SDL correspondante. SDL ne gérant en natif que le format BMP, nous avons choisi pour des raisons de confort d'utiliser cette bibliothèque.

Donc, nous utilisons **SDL_Image** pour charger l'image contenant les sprites dans une surface temporaire **loadedImage**.

```
SDL_Surface* loadedImage=IMG_Load(filename.c_str());
```

Puis, nous demandons à SDL via la fonction **SDL_DisplayFormat** de renvoyer une surface correspondante, mais optimisée pour l'affichage sur l'écran. Cet appel un peu curieux se comprend si on imagine que le format de la surface écran est différent de celui de l'image. Supposons que la surface de l'écran soit au format RGBA et le format de l'image au format RGB.

Dans ce cadre, SDL serait obligé à chaque blit de l'image sur l'écran d'effectuer une conversion de couleur. Via cette fonction **SDL_DisplayFormat**, la conversion est faite une fois pour toutes. La surface convertie devient donc notre planche **spriteMap** et nous pouvons détruire la surface renvoyée par **SDL_FreeSurface**.

```
 this->spriteMap  = SDL_DisplayFormat( loadedImage );
SDL_FreeSurface( loadedImage );
```

Les deux dernières lignes à voir dans cette méthode sont :

```
Uint32 colorkey = SDL_MapRGB( spriteMap->format, 0xFF, 0xFF, 0xFF );
SDL_SetColorKey( spriteMap, SDL_RLEACCEL | SDL_SRCCOLORKEY, colorkey );
```

Il s'agit ici de signaler à SDL quelle est la couleur de transparence pour cette surface. Il s'agit ici du fameux principe de l'écran bleu. Nous définissons une couleur blanche **colorkey** via la fonction **SDL_MapRGB**. Puis, nous précisons à SDL que celle-ci est la couleur de transparence. Comme vu avec l'écran bleu, lors du blit de notre sprite sur un fond d'écran, ne seront recopiés que les pixels non blancs.

Passons maintenant à la méthode **drawMe** qui doit dessiner notre sprite sur une surface. Il s'agit d'un blit qui va dépendre de la direction de notre sprite.

Afin de faire correctement le travail, il faut définir un **SDL_Rect** décrivant la source et la destination pour le blit. Pour la destination et la source, nous connaissons déjà au moins la dimension du blit qui est égale dans les deux cas a **spriteWidth** et **spriteHeight**. Les coordonnées de la destination sont relativement simples, car elles sont données par nos attributs **xpos** et **ypos**.

La seule chose de complexe sont les coordonnées de la source du blit. Celles-ci sont dépendantes de la direction du sprite et sont donc en relation avec l'attribut direction. Nous donnons ici le code de la méthode qui implémente ce que nous avons raconté.

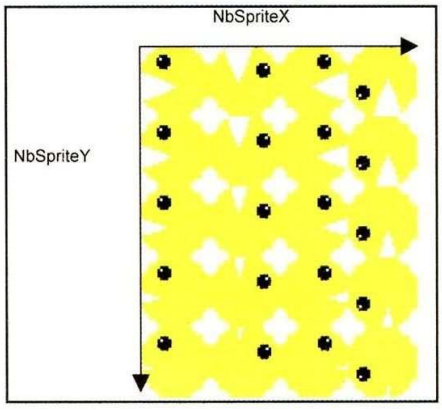
```
void Sprite::drawMe(SDL_Surface * surface)
{
   SDL_Rect rect;
   rect.w=this->spriteWidth;
   rect.h=this->spriteHeight;
   rect.x=(this->direction*this->spriteWidth);
   rect.y=(this->spriteHeight);
   SDL_Rect dst;
   dst.x=xpos;
   dst.y=ypos;
   dst.w=spriteWidth;
   dst.h=spriteHeight;
   SDL_BlitSurface(this->spriteMap, &rect,surface, &dst);
}
```

4.2 La version dynamique du sprite

Avec notre système, nous avons bien un sprite qui oriente sa tête en fonction de la direction. Autrement dit, celui-ci lève la tête quand il faut aller en haut. C'est pas mal, mais ça reste un statique. Disons qu'un vrai Pacman passe son temps à essayer de manger, et que le nôtre a la bouche coincée (encore une opération de lifting ratée :).

Nous allons donc lui donner du dynamisme en lui ajoutant une petite animation qui va lui faire ouvrir et fermer la bouche régulièrement. Une des techniques pour faire de l'animation avec les sprites provient directement des dessins animés. Nous allons découper le mouvement voulu en plusieurs images relativement proches. Pour donner l'illusion du mouvement, il ne restera donc plus qu'à faire défiler « rapidement » les images.

En reprenant notre planche de sprite, nous avons donc horizontalement les images des sprites des différents mouvements (représentés par l'attribut de classe **nbSpriteX**), et verticalement les animations des différents mouvements (modélisés par un nouvel attribut **nbSpriteY**) telles que montrées ici :



Le code de cette fonctionnalité est pratiquement identique au précédent. Récapitulons les changements :

1. Nous rajoutons un attribut **step** qui représente le numéro de l'image de l'animation. Ce numéro est incrémenté à chaque mouvement.

2. Nous ajoutons un attribut **nbSpriteY** dans la classe et dans le constructeur. Cet entier représente le nombre d'images possible pour une animation (ici 5). Le constructeur de la classe devient donc :

```
Sprite(std::string & filename,  unsigned int nbSpriteX,  unsigned int
nbSpriteY, unsigned int spriteWidth, unsigned int spriteHeight);
```

3. Nous devons modifier la fonction **drawMe** qui doit prendre en compte pour le **SDL_Rect** source du numéro de l'image à afficher pour l'animation.

4. Enfin, nous introduisons une fonction membre qui permet de contrôler l'attribut **step** de la classe. En effet, celui-ci doit être modifié à chaque dessin du Pacman afin de prendre en compte l'animation de celui-ci

Nous donnons le code de la nouvelle fonction **drawMe**, ainsi que de la fonction **incrementStep** permettant de contrôler l'attribut **step**.

```
void Sprite::drawMe(SDL_Surface * surface)
{
  SDL_Rect rect;SDL_Rect dst;
  rect.w=this->spriteWidth;  rect.h=this->spriteHeight;
  //Step représente le numéro du sprite pour l'animation. Le seul
changement pour le blit
  // est que l'ordonnée y pour le SDL_rect de la surface source devient
step*spriteHeight
  rect.x=(this->direction*this->spriteWidth);rect.y=(this->step*this-
>spriteHeight);
  dst.x=xpos;dst.y=ypos;dst.w=spriteWidth; dst.h=spriteHeight;
  SDL_BlitSurface(this->spriteMap, &rect,surface, &dst);
}
// incrementStep modifie l'attribut step de la classe
// qui représente le numéro de l'image d'un sprite pour une animation
// ce numéro est compris entre 0 et le nombre d'images possible pour
une animation ici nbSpriteY
// Cette fonction doit être appelée à chaque fois qu'un sprite est dessiné
void Sprite::incrementStep()
{
  if (step<nbSpriteY) { step++; } else { step=0; }
}
```

5 Jouons avec nos sprites

Nous allons maintenant jouer avec notre sprite au sein de la boucle main. En premier lieu, il nous faut construire notre sprite du Pacman via ces deux lignes de codes :

```
std::string pacmanGif("image/pacman.gif");

Sprite pacman(pacmanGif, (unsigned int)5,// la planche de sprite est
composé de
   (unsigned int)4,// 5x4 pacman
   (unsigned int)25, // de 25 pixels de largeur et de
   (unsigned int)25);// longeur chacun
```

Puis, nous devons gérer les interactions utilisateur dans la même structure **switch case** que celle vue plus haut. Les touches utilisées **'f'**,**'g'**,**'t'** et **'v'** demandent au Pacman d'aller respectivement à gauche, à droite, en haut et en bas :

```
case SDL_KEYDOWN:
   /* Check the SDLKey values and move change the coords */
   switch( event.key.keysym.sym ){
   case 'q': exit(0); break;
  case 'f': pacman.changeDirection(0); break;
  case 'g': pacman.changeDirection(2); break;
   case 't': pacman.changeDirection(1); break;
   case 'v': pacman.changeDirection(3); break;
   }
```

Ensuite, en fonction de la touche pressée, nous déplaçons le Pacman si celui-ci ne sort pas des limites de l'écran. Dans ce morceau de code, les variables **SCREEN_WIDTH** et **SCREEN_HEIGHT** représentent respectivement la largeur et la hauteur de l'écran.

```
if (pacman.getDirection()==0 && pacman.getX()>0) {pacman.retrieveX();}
if (pacman.getDirection()==2 && pacman.getX()<SCREEN_WIDTH-25 ){pacman.moveX();}
if (pacman.getDirection()==3 && pacman.getY()<SCREEN_HEIGHT-25){pacman.moveY();}
if (pacman.getDirection()==1 && pacman.getY()>0){pacman.retrieveY();}
```

Il ne reste plus qu'à blitter la surface contenant *le background*, puis le Pacman et, enfin, nous affichons l'écran :

```
SDL_BlitSurface(background, NULL, screen, NULL );
pacman.drawMe(screen);// on dessine le pacman
pacman.incrementStep() ; // et on l'anime
 SDL_Flip( screen ) ;
```

6 La gestion du temps

Lors de l'exécution du programme précédent, on remarquera que le Pacman s'affiche aussi vite que possible. Par « aussi vite », il faut lire que nous allons faire travailler la carte vidéo à son maximum ce qui n'est pas forcément ce qui est voulu car :

- Un jeu peut avoir une vitesse d'affichage correcte (ni trop rapide, ni trop lente) sur une machine et être trop rapide sur une autre.
- Plein d'effets bizarres peuvent se produire si la carte est trop sollicitée. Dans certains des problèmes plus ou moins gênants, on entend le ventilateur de la carte se mettre à siffler, les sprites se mettre à clignoter, ou l'ordinateur s'éteindre à cause d'une surchauffe de la carte vidéo.

Bref, vouloir afficher au maximum possible de la carte est une vraiment mauvaise idée. C'est pour cela que l'on introduit la mesure FPS (*Frame Per Second*) qui est le nombre d'images affichées par seconde souhaité. Lorsqu'on dit qu'un jeu tourne à 25 FPS, on essayera donc d'afficher 25 images par seconde.

Nous introduisons donc la classe **Timer** qui représente un compteur de temps qui va nous permettre de contrôler le nombre d'images par seconde que nous voulons afficher. Ci-joint l'interface de la classe **Timer** commentée :

```
/**
 * La classe Timer représente un compteur de temps
 * nécessaire pour le contrôle du nombre d'images par seconde
 * que l'on souhaite afficher
 */
class Timer
{
   private:
  //Nombre de Ticks depuis le démarrage du compteur
  int startTicks;
  //Nombre d'images par seconde voulu pour l'animation
  int frames_per_second
  bool started;
  public:
  /**
  * Constructeur prenant en paramètre le nombre d'image par seconde du jeu
  */
  Timer(int frames_per_second);
  // Démarre et arrête le compteur
  void start();
  void stop();
  /**
  * renvoie vrai pour signaler que le jeu peut afficher une autre image
  * sans que l'animation dépasse le nombre de FPS voulu
  */
  bool isFramesPerSecondReached();
};
```

Cette classe marche de façon très simple. Notre fonction **main** contient une boucle principale dont le schéma est le suivant :

```
while (true)
{
  // gestion des interactions utilisateur
  // deplacement des sprites
  // affichage sur l'écran
}
```

Afin que la classe **Timer** soit utilisée, il faut créer un objet **T** de cette classe avant le **while(true)**. Puis, le compteur **T** doit être démarré au début de la boucle, et une attente active doit être introduite à la fin via un **while (T.isFramesPerSecondReached());**.

Le tout donne :

```
Timer T(25);
while(true)
{
  T.start();
  // gestion des interactions, déplacement et affichage
  while (T.isFramesPerSecondReached());
}
```

Cela implique qu'une itération de la boucle :

- soit ait pris suffisamment de temps afin que l'affichage ne soit pas trop rapide ; dans ce cas, il n'y a pas d'attente ;
- soit que celle-ci est trop rapide, et nous devons attendre un peu afin de respecter le critère de FPS.

Afin d'implémenter cette classe, nous allons utiliser une série de fonctions fournies par SDL destinées à la gestion du temps. En particulier, nous allons utiliser **SDL_GetTicks()** qui renvoie un entier sur 32 bits représentant le nombre de millisecondes passées depuis l'initialisation de la bibliothèque.

L'implémentation de classe **Timer** devient, avec cette aide, relativement aisée :

```
Timer::Timer(int frames_per_second)
{
   this->startTicks = 0; this->frames_per_second=frames_per_second;
   this->started = false;
}
void Timer::start()
{
   this->started = true; this->startTicks = SDL_GetTicks();
}
void Timer::stop()
{
   this->started = false;
}
bool Timer::isFramesPerSecondReached()
{
   bool result=false;
   if( started == true ) { result=(SDL_GetTicks() - startTicks) < 1000
/ frames_per_second;}
  return result;
}
```

7 Conclusion

Dans ce premier article, nous avons abattu pas mal de travail, et ceci, même si le résultat n'est pas « époustouflant ».

Nous avons vu comment initialiser la bibliothèque SDL, créer quelques surfaces et, en particulier, nous avons introduit la notion de « sprite ». Le code final permet donc de déplacer un petit Pacman animé à l'écran. Dans un prochain article, nous commencerons à regarder les collisions entre un Pacman et :

- un fantôme via un algorithme de *bounding box* ;
- une forme quelconque via une détection de collision par pixel.

Comme le code de cet article est relativement « lourd », je le laisse sur http://www.gnulinuxmag.com/dnl/LM/LM114/.

La programmation graphique est un domaine en soi. Dans cette série, nous l'abordons via les jeux vidéo, mais les mêmes algorithmes et méthodes peuvent être utilisés dans bien d'autres domaines comme la reconnaissance de forme ou le filtrage d'image.

Auteur : p-e-g

Je profite de ces quelques lignes pour souhaiter la bienvenue au monde à la petite Noémie (née en décembre dernier) qui saura, l'heure venue, profiter de cet article pour s'initier à la programmation, au C/C++ ainsi qu'à la bibliothèque SDL.

OCaml et C : le meilleur des

Cet article, le premier d'une série de deux, se propose d'illustrer les techniques de bindings de code C en OCaml avec un exemple pris dans la bibliothèque Ogg. Le but est de montrer comment le binding OCaml permet d'automatiser les tâches bas niveau du programmeur afin de se concentrer, dans le code OCaml, sur l'implémentation à proprement parler et la logique de programmation.

Auteur

- Romain Beauxis

1 Présentation

Lors des différents projets de programmation que l'on peut rencontrer, un des problèmes qui se posent souvent est la possibilité d'implémenter une spécification en vérifiant que le programme respecte bien les contraintes requises.

Plus généralement, le travail de vérification et de correction des bugs d'un programme est souvent grandement simplifié lorsque le programmeur n'a à se concentrer que sur la logique de fonctionnement de son code, et non sur les erreurs « bas-niveau » comme l'allocation et la libération de mémoire.

Or, lorsque la logique de fonctionnement est complexe ou si l'on a à manipuler des objets dynamiques, comme des tableaux à taille variable dans un langage bas niveau comme le C, la lisibilité du code est souvent grandement affectée par toutes les manipulations bas niveau que cela implique. Ainsi, la logique d'implémentation est mélangée avec des tâches plus administratives qui rendent le travail de *debug* bien plus délicat.

Les langages plus récents, comme OCaml, proposent, pour résoudre cela, d'automatiser une partie des tâches bas niveau, comme l'allocation/désallocation en mémoire, qui est gérée par le *garbage collector*, ou l'utilisation d'un système de type riche afin de représenter les contraintes d'implémentation.

Cependant, les manipulations bas niveau sont aussi nécessaires dans le cas d'implémentations moins abstraites, comme la manipulation de flux de bits ou du matériel. De plus, beaucoup de bibliothèques sont fournies sous la forme de fonctions définies en C et ne sont pas directement utilisables en OCaml.

Cet article est le premier d'une série qui se propose d'exposer comment tirer le meilleur des deux niveaux de travail. D'une part, il s'agit d'adapter une bibliothèque en C en montrant comment les tâches administratives peuvent être adaptées pour un traitement par OCaml, puis, ensuite comment utiliser l'interface haut-niveau ainsi disponible pour implémenter une spécification complexe.

Nous choisissons ici de présenter une partie de l'interface de programmation de la bibliothèque Ogg. En effet, les contraintes d'implémentation d'un flux Ogg contenant plusieurs flux audio et vidéo séquentialisés sont complexes. Ainsi, les programmes implémentant cette spécification en C sont-ils souvent difficiles à lire.

Nous nous proposons dans cette première partie de montrer un exemple simple d'adaptation de la bibliothèque Ogg pour utilisation dans du code OCaml. Ensuite, dans un prochain article, nous verrons comment l'utiliser pour implémenter un générateur simple de flux Ogg pour lequel nous n'aurons plus qu'à écrire la spécification Ogg.

Ainsi, le programmeur est confronté à deux tâches bien distinctes : d'un côté, les manipulations bas niveau, de l'autre, la logique d'implémentation, pour lesquelles il peut se concentrer séparément et ainsi être plus attentif aux possibles erreurs de part et d'autre.

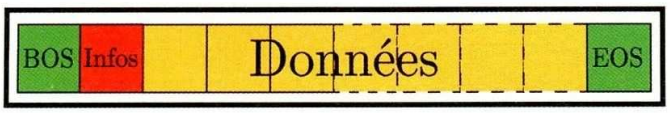
2 Structure d'un flux logique Ogg

Un flux Ogg est composé de plusieurs flux logiques. Chacun des flux logiques contient des données audio, vidéo ou autre, comme des sous-titres. Les flux logiques sont multiplexés,

c'est-à-dire qu'ils sont combinés entre eux pour former un seul flux physique, qui sera en fait le fichier Ogg complet.

Nous nous intéressons dans cet article à la génération d'un flux logique Ogg. Les contraintes relatives à leur multiplexage seront vues ultérieurement. Dans la suite de cette partie, nous décrivons la structure d'un tel flux logique. La documentation sur le sujet peut être consultée sur le site de la fondation Xiph **[I]**.

2.1 Flux logique Ogg



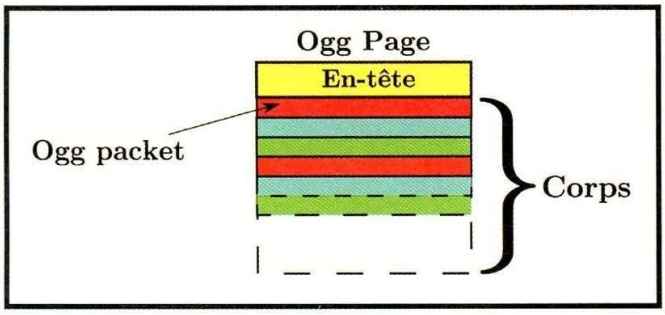
Un flux logique Ogg est formé d'une succession d'éléments appelés « pages ». Chaque page contient un groupe de paquets qui sont les éléments unitaires du flux sous-jacent. Ainsi, lors du décodage du flux, les pages sont lues une par une, puis découpées en paquets qui sont ensuite passés un par un au décodeur correspondant.

Un flux logique doit impérativement commencer par une page d'en-tête ou *begining of stream* (BOS) contenant les informations sur le flux sous-jacent. Par exemple, dans le cas d'un flux vorbis, celle-ci contiendra l'en-tête spécifique au codec vorbis, ainsi que diverses informations importantes pour le décodeur.

Ensuite, toujours dans le cas d'un flux vorbis, les premiers paquets de la seconde page contiendront les informations sur le morceau (métadata), comme son titre, l'auteur, etc. Il est recommandé dans les contraintes de flux que cette deuxième page ne contienne pas de données audio ou vidéo, de telle sorte que, lorsque les flux seront multiplexés, toutes les données commencent en même temps. Ainsi, cette page est-elle en général obtenue à l'aide de la fonction `ogg_stream_flush`.

S'ensuivent une succession de pages contenant les données du flux. Enfin, une fois le flux terminé, la dernière page doit mentionner l'information de fin de flux ou *end of stream* (EOS).

2.2 Pages Ogg



Une page Ogg est composée de deux parties : un en-tête, suivi d'un corps contenant ses données. Les informations de début et de fin de flux sont contenues dans l'en-tête, les paquets de données dans son corps.

Sa déclaration en C est :

```
typedef struct {
  unsigned char *header;
  long header_len;
  unsigned char *body;
  long body_len;
} ogg_page;
```

Afin de les manipuler, la bibliothèque Ogg propose diverses fonctions, comme :

- `ogg_page_bos` : détermine si la page est une page de début de flux.
- `ogg_page_eos` : détermine si la page est une page de fin de flux.
- `ogg_page_serialno` : retourne le numéro de série d'une page. Très important afin de traiter des pages venant de différents flux logiques lors du multiplexage.
- `ogg_page_packets` : retourne le nombre de paquets contenus dans la page.

Ainsi que d'autres que nous ne détaillons pas ici.

2.3 Paquets Ogg

Un paquet Ogg contient les données du flux, ainsi que diverses informations. En particulier, les informations de début et fin de flux. Lors de l'encodage, une fois un paquet contenant le marqueur de début de flux reçu, la page de début de flux est alors produite. De même pour la page de fin de flux.

La déclaration d'un paquet Ogg en C est :

```
typedef struct {
  unsigned char *packet;
  long  bytes;
  long  b_o_s;
  long  e_o_s;

  ogg_int64_t  granulepos;

  ogg_int64_t  packetno;
} ogg_packet;
```

Les informations `granulepos` et `packetno` sont propres au flux sous-jacent. `packetno` désigne le numéro du paquet dans la séquence. Ceux-ci doivent se suivre. S'il en manque un, le décodeur saura alors que le flux est tronqué. `granulepos` est une donnée propre au décodeur. Elle désigne la position courante au sein du flux encodé. Par exemple, pour un flux vorbis, elle désigne la position en *samples* audio dans le flux.

La bibliothèque Ogg ne propose pas de fonctions de manipulation des paquets Ogg. D'une part, ceux-ci sont en fait générés par la bibliothèque d'encodage et, d'autre part, la déclaration en C suffit à manipuler les informations nécessaires, comme les marqueurs de début et de fin de flux.

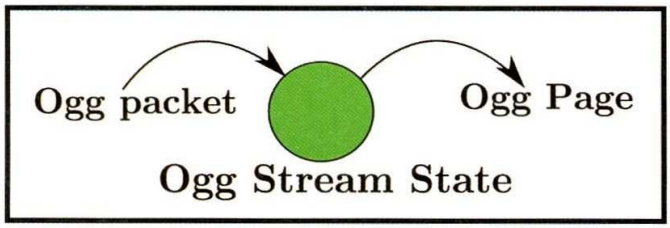
Ainsi, un paquet de fin de flux peut être forgé comme suit :

```
ogg_packet op;

/* Forge un paquet de fin de flux */
op.packet = NULL;
op.bytes = 0;
op.b_o_s = 0;
op.e_o_s = 1; /* Passe le marqueur de fin de flux. */
op.granulepos = granulepos;
op.packetno = packetno;
```

où les valeurs **granulepos** et **packetno** sont celles du dernier paquet, incrémentées d'une unité.

2.3.1 Stream state



Dans la bibliothèque de manipulation des flux Ogg, un état de flux (*stream state*) est un objet abstrait permettant de forger les pages d'un flux à partir de ses paquets ou l'inverse, c'est-à-dire de découper la suite des pages d'un flux en paquets de données qui sont ensuite passés au décodeur.

Sa déclaration en C n'est pas mentionnée ici, car elle contient beaucoup d'informations internes dont nous ne ferons pas usage. En effet, les fonctions de manipulation de l'état de flux dans la bibliothèque Ogg sont variées et permettent de travailler sans jamais avoir à regarder le contenu d'un tel animal.

Les fonctions qui nous intéressent ici sont :

- **ogg_stream_packetin** : ajoute un paquet au flux courant.
- **ogg_stream_pageout** : retourne, si suffisamment de paquets ont été passés, une page du flux courant. Cette fonction ne retourne pas toujours une page. En revanche, une page est immédiatement retournée si le dernier paquet passé avait un marqueur de début ou fin de flux.
- **ogg_stream_flush** : force la création d'une page contenant les paquets du flux. Cette fonction n'est pas nécessaire en général. Cependant, elle est par exemple utilisée pour produire la seconde page du flux, puisqu'on a vu que, idéalement, celle-ci ne contient que les méta-données du flux. Une fois les paquets contenant ces informations passés au flux, on appellera donc cette fonction pour forcer la seconde page à ne contenir que ces paquets.

Un état de flux fonctionnant dans les deux sens, il existe des fonctions symétriques, **ogg_stream_pagein**, **ogg_stream_packetout** et **ogg_stream_packetpeek** que nous ne traiterons pas ici.

Enfin, un état de flux est initialisé avec la fonction **ogg_stream_init**, réinitialisé avec la fonction **ogg_stream_reset** et nettoyé avec la fonction **ogg_stream_destroy**. La fonction **ogg_stream_eos** permet de savoir si le flux logique est terminé. D'autres fonctions, que nous ne détaillons pas ici, existent. La référence en ligne de la bibliothèque Ogg **[2]** est très complète sur ce sujet.

3 Interfacer le C et OCaml

Le sujet de cette partie est d'évoquer par l'exemple les techniques qui permettent d'utiliser des bibliothèques écrites en C dans du code écrit en OCaml. Le principe consiste à écrire un code minimal en C faisant l'interface avec les fonctions de la bibliothèque C, puis à récupérer les fonctions ainsi exportées dans un fichier OCaml. En termes techniques, cela s'appelle un *binding*.

Nous allons, dans un premier temps, expliquer certaines choses simples sur les différents types de données en C, leurs équivalents en OCaml et, enfin, comment adapter un type de données C pour du code OCaml. Ensuite, nous verrons comment déclarer les primitives C permettant d'adapter les fonctions de la bibliothèque C à du code OCaml. Enfin, nous verrons comment définir de nouveaux types abstraits en OCaml et les utiliser pour manipuler des objets C, puis, pour finir, comment exploiter ces primitives dans le code OCaml associé.

Les fonctions de conversion les plus usuelles sont contenues dans les fichiers d'en-tête placés dans **/usr/include/caml** ou **/usr/local/include/caml** si vous avez compilé vous-même OCaml. Les conversions sont fournies soit sous la forme d'une macro C, soit sous celle d'une vraie fonction.

Une documentation est disponible en ligne **[3]**. Nous nous contentons ici de donner les grandes lignes sur le sujet. Le lecteur intéressé est invité à consulter les autres sources d'informations. Enfin, le projet Savonet contient une longue liste d'exemples de bindings **[4]** pour diverses bibliothèques plus ou moins simples, qui peuvent aussi être consultées pour voir des exemples concrets.

3.1 Les types de données

Lors de l'utilisation d'une bibliothèque C depuis du code OCaml, à chaque appel d'une fonction de la bibliothèque avec des arguments venant du code OCaml, il faut :

- convertir les arguments OCaml en arguments C ;
- appliquer la fonction C ;

- convertir de résultat en argument Ocaml ;

Les types de données les plus communs sont alors :

- les entiers ;
- les booléens ;
- les chaînes de caractères ;
- les objets abstraits.

Les valeurs OCaml sont représentées dans du code C par le type `value`. Le cas des objets abstraits sera traité dans la prochaine section. Nous détaillons ici les autres types.

3.1.1 Les entiers

Pour le cas des entiers, les entiers OCaml sont répartis en plusieurs types :

- `Int` : représentation la plus simple et la plus rapide. La représentation en mémoire dépend de l'architecture. Sur une architecture 32 bits, elle sera de 31 bits (un bit est utilisé par le garbage collector). Si l'entier que vous souhaitez communiquer depuis le code C à du code OCaml est susceptible d'être plus grand que 31 bits sur une architecture 32 bits, alors vous devrez utiliser un des types suivants. Le même raisonnement s'applique pour les autres architectures.
- `Nativeint.t` ou `nativeint` : entiers dont la taille est exactement celle de l'architecture sous-jacente. La représentation en mémoire est exactement celle d'un type C `long` sur l'architecture sous-jacente.
- `Int32.t` : entiers de taille 32 bits.
- `Int64.t` : entiers de taille 64 bits.

Les fonctions les plus usuelles pour ces conversions sont :

- `Val_int(x)` : macro transformant l'entier C `x` en entier OCaml de type `int`.
- `Int_val(x)` : macro transformant l'entier OCaml `x` en entier C.
- `caml_copy_nativeint` : fonction retournant un entier OCaml `Nativeint.t` construit à partir d'un entier C.
- etc.

3.1.2 Les booléens

Le type booléen OCaml `bool` est converti en C en un entier, dont, à l'image d'un test booléen en C, une valeur non nulle signifie `true`. Les fonctions de manipulation sont :

- `Val_bool(x)` : macro transformant l'entier C `x` en booléen Ocaml.
- `Bool_val(x)` : macro transformant le booléen OCaml en entier C.
- `Val_true`, `Val_false` : macro représentant en C les valeurs OCaml `true` et `false`.

3.1.3 Les chaînes de caractères

Les chaînes de caractères ou `string` sont représentées en C par une suite de caractères terminée par un caractère spécial. Cette représentation est en fait purement locale, puisqu'une chaîne de caractères n'est manipulable que via le caractère courant. Ainsi, pour connaître sa longueur, il faut parcourir les caractères un par un jusqu'à tomber sur le caractère de terminaison.

Au contraire, en OCaml, une chaîne de caractères est définie globalement comme un bloc de mémoire alloué contenant des caractères. En particulier, une `string` OCaml peut éventuellement contenir des caractères après le caractère de terminaison. Ainsi, une `string` OCaml sera souvent une représentation pratique pour des blocs de données venant du C, comme un échantillonnage de données audio PCM à encoder.

Les fonctions disponibles pour les conversions de chaînes de caractères sont :

- `String_val(x)` : macro renvoyant l'adresse C du premier caractère de la chaîne OCaml `x`. Permet de passer une chaîne OCaml à une fonction C sans la copier. Attention cependant, cela peut poser des problèmes si la chaîne OCaml est nettoyée par ailleurs par le garbage collector.
- `caml_copy_string(x)` : fonction retournant une chaîne OCaml contenant une copie de la chaîne C `x`.

3.2 Primitives OCaml en C

Afin d'utiliser les fonctions de votre bibliothèque C, il convient de préparer des fonctions qui vont donc convertir les arguments OCaml en arguments C, puis appliquer la ou les fonctions désirées et, enfin, convertir le résultat et le renvoyer au code OCaml.

Comme OCaml possède un garbage collector, ces fonctions doivent aussi communiquer avec celui-ci afin qu'il sache quels arguments sont en cours d'utilisation et qu'ils ne soient pas nettoyés. Pour cela, des macros C sont définies par les fichiers d'en-tête OCaml.

Ainsi, le code d'une fonction sera de la forme :

```
CAMLprim value ocaml_ogg_stream_serialno(value o_stream_state)
{
  CAMLparam1(o_stream_state);
  ogg_stream_state *os = Stream_state_val(o_stream_state);

  CAMLreturn(caml_copy_nativeint((intnat)os->serialno));
}
```

Cette fonction prend un paramètre de type stream state, récupère le pointeur C correspondant en utilisant la macro `Stream_state_val` (voir section suivante) et, enfin, retourne une représentation `Nativeint.t` de l'entier représentant en C le numéro de série du stream state.

`CAMLprim` est une macro qui est utilisée pour introduire des fonctions manipulant des données OCaml en C. Elle sert à prévenir le garbage collector qu'un traitement en C va avoir lieu. `CAMLparam1` est une macro permettant de définir quels sont les paramètres OCaml que le garbage collector doit préserver. Par exemple, un booléen ne nécessite pas d'être déclaré via cette macro. Si des paramètres locaux

représentant des objets OCaml sont utilisés et que l'on souhaite de même les protéger du garbage collector, il faut alors les déclarer avec une macro de la forme : `CAMLlocal1(x)`. Enfin, la primitive `CAMLreturn` replace le `return` du C, et permet d'informer le garbage collector que le traitement en C a pris fin.

3.3 Les blocs personnalisés

Lors de la préparation de la bibliothèque C pour son usage dans du code OCaml, il arrive aussi que certains objets définis par la bibliothèque n'aient pas de représentation équivalente en OCaml ou alors que l'on souhaite les exporter comme des « boites noires », que l'utilisateur OCaml utilise sans devoir connaître les détails internes de fonctionnement.

Ainsi, par exemple, dans le cas de la bibliothèque Ogg, nous ne souhaiterons pas exporter les détails de l'objet abstrait stream state. Nous allons alors définir un type abstrait dans le code OCaml qui représentera cet objet. D'autre part, dans les primitives C adaptant la bibliothèque, nous devons définir les fonctions permettant d'instancier cet objet, puis de le détruire quand l'utilisateur OCaml a fini de le manipuler.

Ceci est obtenu en utilisant des blocs personnalisés (*custom blocks*). Ces derniers permettent de définir un objet OCaml (donc de type `value` dans le code C), qui contient un pointeur vers l'objet C que l'on souhaite encapsuler, ainsi qu'un autre pointeur vers une structure représentant les fonctions par défaut à appliquer sur cet objet abstrait, comme une méthode pour nettoyer l'objet, le comparer avec un autre, etc.

Tout d'abord, on définit une macro qui convertit la représentation OCaml de type `value` en pointeur C vers l'objet sous-jacent :

```
#define Stream_state_val(v) (*((ogg_stream_state**)Data_custom_val(v)))
```

Ensuite, on écrit la fonction qui sera appelée par le garbage collector lors de la destruction de l'objet, ainsi que la structure associée (on laisse les valeurs par défaut pour les autres fonctions) :

```
static void finalize_stream_state(value s)
{
  // This also free the argument
  ogg_stream_destroy(Stream_state_val(s));
}

static struct custom_operations stream_state_ops =
{
  "ocaml_ogg_stream_state",
  finalize_stream_state,
  custom_compare_default,
  custom_hash_default,
  custom_serialize_default,
  custom_deserialize_default
};
```

Pour finir, on définit une fonction qui initialise un tel objet :

```
CAMLprim value ocaml_ogg_stream_init(value serial)
{
  CAMLparam0();
  CAMLlocal1(ans);
  ogg_stream_state *os = malloc(sizeof(ogg_stream_state));

  ogg_stream_init(os, Nativeint_val(serial));
  ans = caml_alloc_custom(&stream_state_ops, sizeof(ogg_stream_state*),
1, 0);
  Stream_state_val(ans) = os;

  CAMLreturn(ans);
}
```

Une fois intégré dans le code OCaml, on aura alors la possibilité d'instancier un objet abstrait représentant un stream state de la bibliothèque Ogg, puis de l'utiliser. Un avantage pour l'utilisateur OCaml sera alors que le garbage collector OCaml saura exactement quand et comment nettoyer cet objet lorsqu'il n'est plus utilisé, évitant les fuites de mémoire.

3.4 Déclaration en OCaml

La déclaration en OCaml des fonctions définies dans les primitives C est assez simple. Ainsi, par exemple, pour déclarer le type abstrait stream state, ainsi que la fonction d'initialisation, on écrira dans un fichier `ogg.ml` :

```
type stream_state

external stream_init : nativeint -> stream_state = "ocaml_ogg_stream_
init"
```

De plus, il est aussi conseillé d'exporter une interface de programmation `ogg.mli` comme suit :

```
type stream_state

val stream_init : nativeint -> stream_state
```

En effet, la redéclaration comme type `val` dans le fichier `ogg.mli` évite quelques bugs connus introduits par l'utilisation du mot-clef `external` **[5]**.

4 Application

Dans cette partie, nous définissons une interface OCaml pour la création de flux Ogg. Le code est volontairement simplifié afin de garder un aspect didactique. Le binding complet `ocaml-ogg` a été publié par le projet Savonet **[6]**. Le lecteur intéressé pourra le consulter pour plus de détails.

4.1 Fonctions en C

La fonction d'initialisation d'un stream state a été présentée plus haut. Nous définissons maintenant une

représentation OCaml pour les pages et les paquets Ogg :

Pour les paquets Ogg, on choisit une représentation par un bloc personnalisé contenant une copie du paquet d'origine. Cela donne :

```
/* Macro pour extraire un paquet ogg d'un block personalisé */
#define Packet_val(v) (*((ogg_packet**)Data_custom_val(v)))

/* fonction de nettoyage d'un paquet Ogg */
static void finalize_packet(value s)
{
  ogg_packet *op = Packet_val(s);
  free(op->packet);
  free(op);
}

/* Structure contenant les opérations par défaut sur un paquet. */
static struct custom_operations packet_ops =
{
  "ocaml_ogg_packet",
  finalize_packet,
  custom_compare_default,
  custom_hash_default,
  custom_serialize_default,
  custom_deserialize_default
};

/* Fonction qui alloue un nouveau paquet ogg et place dedans
 * les données du paquet passé en paramètre. */
static inline ogg_packet *copy_packet(ogg_packet *op)
{
  ogg_packet *nop = malloc(sizeof(ogg_packet));
  if (nop == NULL)
    caml_failwith("malloc");
  nop->packet = malloc(op->bytes);
  memcpy(nop->packet,op->packet,op->bytes);
  nop->bytes = op->bytes;
  nop->b_o_s = op->b_o_s;
  nop->e_o_s = op->e_o_s;
  nop->granulepos = op->granulepos;
  nop->packetno = op->packetno;

  return nop;
}

/* Fonction qui crée un bloc personalisé contenant une copie d'un
paquet Ogg. */
CAMLprim value value_of_packet(ogg_packet *op)
{
  CAMLparam0();
  CAMLlocal1(packet);
  packet = caml_alloc_custom(&packet_ops, sizeof(ogg_packet*), 1, 0);
  Packet_val(packet) = copy_packet(op);
  CAMLreturn(packet);
}
```

Pour les pages, on va utiliser une représentation sous la forme d'un couple de chaînes de caractères OCaml. Le premier élément représentera l'en-tête, le second le corps de la page. Cette représentation est bien plus commode à manipuler, en particulier pour écrire les pages dans le fichier Ogg final. Elle nécessite cependant de copier les données de la page.

Cela donne :

```
CAMLprim value value_of_page(ogg_page *op)
{
  CAMLparam0();
  CAMLlocal3(v,header,body);
  header = caml_alloc_string(op->header_len);
  memcpy(String_val(header),op->header,op->header_len);

  body = caml_alloc_string(op->body_len);
  memcpy(String_val(body),op->body,op->body_len);

  /* La fonction caml_alloc_tuple alloue
   * un couple de données OCaml. */
  v = caml_alloc_tuple(2);
  /* On place le header dans la permière valeur. */
  Store_field(v,0,header);
  /* On place le corps dans la seconde. */
  Store_field(v,1,body);

  CAMLreturn(v);
}
```

Enfin, on définit une fonction ajoutant un paquet au flux courant. **unit** est équivalent au type **void** en C, c'est-à-dire le résultat vide. **Val_unit** est une macro C représentant la valeur **unit**.

```
CAMLprim value ocaml_ogg_stream_packetin(value o_stream_state, value
packet)
{
  CAMLparam2(o_stream_state, packet);
  ogg_stream_state *os = Stream_state_val(o_stream_state);

  if (ogg_stream_packetin(os, Packet_val(packet)) != 0)
    /* Cette constante est définie dans le code OCaml. */
    caml_raise_constant(*caml_named_value("ogg_exn_bad_data"));

  CAMLreturn(Val_unit);
}
```

Une fonction retournant une page, si elle existe :

```
CAMLprim value ocaml_ogg_stream_pageout(value o_stream_state)
{
  CAMLparam1(o_stream_state);
  ogg_stream_state *os = Stream_state_val(o_stream_state);
  ogg_page og;

  if(!ogg_stream_pageout(os, &og))
    /* Cette constante est définie dans le code OCaml. */
    caml_raise_constant(*caml_named_value("ogg_exn_not_enough_data"));

  CAMLreturn(value_of_page(&og));
}
```

Et, enfin, une fonction exécutant la méthode **ogg_stream_flush**, forçant la production d'une page :

```
CAMLprim value ocaml_ogg_flush_stream(value o_stream_state)
{
  CAMLparam1(o_stream_state);
  ogg_stream_state *os = Stream_state_val(o_stream_state);
  ogg_page og;

  if(!ogg_stream_flush(os, &og))
    /* Cette constante est définie dans le code OCaml. */
    caml_raise_constant(*caml_named_value("ogg_exn_not_enough_data"));

  CAMLreturn(value_of_page(&og));
}
```

4.2 Déclaration OCaml

Le code OCaml du fichier **ogg.ml** correspondant devra tout d'abord déclarer les primitives définies dans le code C.

Cela donne :

```
(* Type représentant un état de flux courant. *)
type stream_state

(* Type représentant un paquet Ogg. *)
type packet

(* Type d'une page Ogg. *)
type page = string*string

(* Initialise un état de flux. *)
external stream_init : nativeint -> stream_state = "ocaml_ogg_stream_
init"

(* Retourne le numéro de série d'un flux. *)
external stream_serial : stream_state -> nativeint = "ocaml_ogg_stream_
serialno"

(* Ajoute un paquet au flux courant. *)
external packetin : stream_state -> packet -> unit = "ocaml_ogg_stream_
packetin"

(* Récupère une page si existante. *)
external pageout : stream_state -> page = "ocaml_ogg_stream_pageout"

(* Force la création d'une page, si possible. *)
external flush : stream_state -> page = "ocaml_ogg_flush_stream"
```

De plus, les primitives en C utilisent, en cas d'échec, une exception. Celle-ci doit alors être définie dans le code OCaml. Cela donne :

```
(* Exception levée quand il n'y a pas assez de données disponibles. *)
exception Not_enough_data
(* Exception levée quand les données sont invalides. *)
exception Bad_data

(* Enregistrement des exceptions pour utilisation en C. *)
let () =
  Callback.register_exception "ogg_exn_not_enough_data" Not_enough_data
;
  Callback.register_exception "ogg_exn_bad_data" Bad_data
```

Enfin, on peut alors définir des fonctions plus haut niveau, par exemple une fonction produisant la séquence maximale de pages, soit normalement, soit en forçant la création :

```
  (* Retourne une string contenant la concaténation
   * de toutes les pages crées. *)
  let pageout_all s =
    let rec f v =
      try
        let (h,b) = pageout s in
        f (v ^ h ^ b)
      with
        | Not_enough_data -> v
    in
    f ""

  (* Retourne une string contenant la concaténation
```

```
  * de toutes les pages crées. *)
 let flush_all s =
   let rec f v =
     try
       let (h,b) = flush s in
       f (v ^ h ^ b)
     with
       | Not_enough_data -> v
   in
   f ""
```

Le fichier **ogg.mli** contient alors :

```
(* Exception levée quand il n'y a pas assez de données disponibles. *)
exception Not_enough_data
(* Exception levée quand les données sont invalides. *)
exception Bad_data

(* Type représentant un état de flux courant. *)
type stream_state

(* Type représentant un paquet Ogg. *)
type packet

(* Type d'une page Ogg. *)
type page = string*string

(* Initialise un état de flux. *)
val stream_init : nativeint -> stream_state

(* Retourne le numéro de série d'un flux. *)
val stream_serial : stream_state -> nativeint

(* Ajoute un paquet au flux courant. *)
val packetin : stream_state -> packet -> unit

(* Récupère une page si existante. *)
val pageout : stream_state -> page

(* Force la création d'une page, si possible. *)
val flush : stream_state -> page

(* Retourne la concaténation de toutes les pages qui
 * peuvent être crées en utilisant pageout. *)
val pageout_all : stream_state -> string

(* Retourne la concaténation de toutes les pages qui
 * peuvent être crées en utilisant flush. *)
val flush_all : stream_state -> string
```

4.3 Compilation

La compilation se fait à l'aide des commandes suivantes :

```
ocamlc -c ogg_stubs.c
ar rcs libogg_stubs.a ogg_stubs.o
ocamlc -c ogg.mli
ocamlc -c ogg.ml
ocamlmklib -o ogg_stubs  ogg_stubs.o -logg
ocamlc -a -dllib dllogg_stubs.so -custom -cclib -logg_stubs -cclib
-logg -o ogg.cma ogg.cmo
```

Un meilleur support serait obtenu en utilisant **OCamlMakefile**, ainsi que **ocaml-findlib**, mais cela dépasse le cadre de cet article.

Enfin, nous pouvons compiler le code trivial suivant, contenu dans le fichier **test.ml** :

```
let serial = Nativeint.of_int 1234

let state = Ogg.stream_init serial
Printf.printf "Un état de flux a été crée avec le numéro de série:
%ni\n%!"
  (Ogg.stream_serial state)
```

Cela donne :

```
19:20 toots@leonard ~/ocaml-ogg-demo% ocamlc ogg.cma ogg.cmo test.ml -I
. -o test
19:20 toots@leonard ~/ocaml-ogg-demo% ./test
Un état de flux a été crée avec le numéro de série: 1234
```

Les fichiers de code sont disponibles en ligne **[7]**.

5 Conclusion

Nous avons vu ici comment adapter les tâches bas niveau d'une partie de la bibliothèque Ogg afin de pouvoir l'utiliser dans du code OCaml. Ainsi, les problèmes d'allocation/désallocation mémoire peuvent être soigneusement vérifiés à un seul endroit dans le code. De plus, la création et la manipulation de ces objets sera grandement simplifiée dans l'utilisation ultérieure dans du code OCaml, permettant de se concentrer sur la logique de programmation.

Nous verrons dans le prochain article comment tirer parti de cette interface pour implémenter une partie de la spécification Ogg afin de produire des flux Ogg corrects.

Références

[1] http://xiph.org/ogg/doc/

[2] http://xiph.org/ogg/doc/libogg/reference.html

[3] http://caml.inria.fr/pub/docs/manual-ocaml/manual032.html

[4] http://savonet.rastageeks.org/browser/trunk

[5] http://caml.inria.fr/mantis/view.php?id=4166

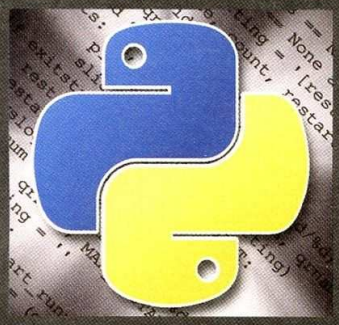
[6] http://savonet.sf.net/

[7] http://www.rastageeks.org/~toots/ocaml-ogg-demo/

Auteur : Romain Beauxis

Utilisateur GNU/Linux depuis 2004, développeur pour le projet Savonet

Python pour la publication



Auteur

- Christophe Combelles

Dans le domaine du web, le langage Python est resté jusqu'ici assez discret, bien qu'il soit intensivement utilisé depuis une dizaine d'années. Sa discrétion est principalement due à trois facteurs : tout d'abord, il y a peu de moyens de savoir qu'un site fonctionne avec Python. L'exposition directe des fichiers PHP par les serveurs web sous la forme d'URL se terminant par « .php » a été une source importante de visibilité dont profite rarement Python. D'autre part, les questions du déploiement ont joué un rôle important : bien qu'il soit en théorie possible, le déploiement par simple glisser-déposer dans un client FTP n'est pas la méthode la plus courante et mettre en place une application Python nécessite souvent un accès au shell, chose rare dans les hébergements mutualisés. La raison principale est que la majorité des applications Python, comme les applications Java, sont des long-running processes, c'est-à-dire un processus permanent, distinct du serveur web principal. Enfin, le Python étant avant tout le fruit d'une communauté de personnes et d'entreprises, il ne bénéficie pas des efforts marketing d'une entité unique essayant d'imposer sa technologie.

Mais les choses ont évolué : trois autres facteurs décisifs ont contribué à faire aujourd'hui de Python une plateforme idéale et rassurante pour le web : les qualités intrinsèques de ce langage, sa concision, sa lisibilité, la souplesse de ses types de données, son énorme bibliothèque standard, sa prise en charge de l'Unicode et des namespaces, ses outils de tests unitaires, vous feront gagner du temps et vous aideront à écrire des applications fiables et pérennes. Ensuite, la diversité des *frameworks* web vous offre un vaste choix pour couvrir tous les besoins, depuis la page perso jusqu'à l'application de gestion d'un grand compte, en passant par le dernier site web 2.0 à fort trafic (YouTube est écrit presque entièrement en Python !). Cette diversité est heureusement canalisée par une norme, le WSGI, qui rend tous les frameworks Python interopérables et permet l'insertion de *middlewares* pour traiter des choses aussi diverses que l'authentification ou l'habillage. Dernier point, la baisse du coût des hébergements privés et dédiés, la virtualisation et les initiatives de *cloud computing* offrent des solutions à la fois abordables et performantes pour vos applications Python.

Nous allons tout d'abord vous donnez un aperçu des différents modes de déploiement de Python pour le web, tester chacune des méthodes en commençant avec le plus simple et le plus vieux, le CGI. Ensuite, nous aborderons trois frameworks modernes ayant chacun leur propre philosophie. Les exemples suivants sont exécutés sur une Debian Lenny, avec Apache 2.2 en mode *mpm-worker*. Le minimum à installer se résume avec la commande suivante :

```
# aptitude install apache2 build-essential python
python-dev python-setuptools python-virtualenv
```

1 CGI

Le CGI a été la première façon de déployer une application web et on le rencontre encore aujourd'hui occasionnellement : il est possible de mettre en ligne un wiki MoinMoin ou l'interface web d'un dépôt Mercurial grâce au CGI. Son principal intérêt est sa simplicité, sa rapidité de mise en œuvre et son caractère universel, mais ses performances sont exécrables, car un nouveau processus est lancé à chaque requête. Il est également plus difficile de

conserver des informations entre deux requêtes comme des sessions ou des connexions à des bases de données.

Selon le mode de fonctionnement d'Apache (*multithread* ou *multiprocess*), c'est le module **mod_cgi** ou **mod_cgid** qui sera utilisé. Ces deux modules fonctionnent de la même façon, et devraient déjà être installés. Il vous suffit de lancer **a2enmod** cgi pour choisir et activer automatiquement le bon module, puis de lancer **/etc/init.d/apache2 restart** pour redémarrer le serveur. Votre configuration Apache devrait déjà contenir une section pour pouvoir exécuter des scripts CGI depuis le répertoire **/usr/lib/cgi-bin**. Il ne reste plus qu'à écrire notre « *Hello World* » dans le fichier **/usr/lib/cgi-bin/hello_cgi.py** :

```
#!/usr/bin/python
print 'Content-Type: text/html'
print ''
print '<html><body>hello</body></html>'
```

Rendez ce script exécutable : **chmod +x /usr/lib/cgi-bin/hello_cgi.py**, puis vous pouvez voir apparaître le hello à l'adresse http://localhost/cgi-bin/hello_cgi.py. Ce script se contente de renvoyer le strict nécessaire pour que votre navigateur affiche quelque chose : une ligne d'en-tête HTTP pour donner le type de contenu, puis une ligne vide pour la séparation, puis une ligne de contenu en HTML.

Python fournit par défaut un module nommé **cgi** qui contient des outils et des classes, par exemple pour parcourir un formulaire HTML (**cgi.FieldStorage**). Un autre module (**cgitb**) permet d'afficher les erreurs Python directement dans le navigateur : pour l'activer, placez la ligne **import cgitb; cgitb.enable()** au tout début du script.

2 mod_python

Comme pour le PHP avec **mod_php**, un module Apache a été développé pour permettre d'embarquer l'interprète Python dans le serveur web. **mod_python** est un module très évolué qui permet d'accéder à de nombreux paramètres internes d'Apache. Toutes les phases de traitement d'une requête par Apache peuvent être écrites en Python et personnalisées. Il vous offre également le choix entre trois gestionnaires de requête (les *handlers*) : un émulateur de CGI, un publicateur inspiré de celui de Zope 2 qui route les URL vers une variable ou une fonction, et enfin un handler appelé PSP (*Python Server Pages*) qui autorise comme pour PHP, ASP ou JSP d'exécuter du code Python se trouvant dans une page HTML. Il est même possible d'exécuter du Python en mode SSI (*Server Side Includes*).

Toutefois, il est généralement déconseillé d'utiliser **mod_python**, car ce module souffre d'un certain nombre d'inconvénients qui sont inhérents au fait d'embarquer l'interprète dans le serveur web (lien fort avec une version de Python, chargement dans tous les processus fils, etc.). Python est un langage polyvalent qui offre des possibilités d'utilisation très vastes. Grâce à PSP, vous pouvez reproduire si vous le souhaitez les mêmes méthodes qu'en PHP, mais, en pratique, personne ne procède de la sorte : les nombreuses alternatives disponibles sont généralement plus propres, plus souples et plus intéressantes.

Malgré tout, nous allons créer un petit exemple pour chacune des possibilités offertes par **mod_python**. Il faut d'abord installer et activer **mod_python**, ce qui, sur un système Debian, se fait de la façon suivante :

```
# aptitude install libapache2-mod-python
# a2enmod python
# /etc/init.d/apache2 restart
```

2.1 Un handler minimaliste

Voici tout d'abord un exemple du plus petit handler possible. Il faut d'abord configurer Apache. Nous allons demander à **mod_python** de prendre en charge toutes les URL qui démarrent avec **/test_mod_python/** et d'utiliser un handler nommé **mon_handler** qui est un simple module Python.

```
<Location "/test_mod_python/">
    SetHandler mod_python
    PythonHandler mon_handler
    PythonPath "['/var/www/monpackage/'] + sys.path"
</Location>
```

Voici le handler, créé dans le fichier **/var/www/monpackage/mon_handler.py** :

```
from mod_python import apache

def handler(req):
    req.content_type = "text/html"
    req.write("<html><body>hello</body></html>")
    return apache.OK
```

Vous pouvez ensuite observer le « hello » à l'adresse http://localhost/test_mod_python/. Remarquez dès la première ligne que vous disposez d'un module Apache qui vous donne accès aux paramètres internes du serveur.

2.2 Émulateur CGI

Nous allons maintenant remplacer notre handler minimaliste par l'émulateur CGI fourni avec **mod_python**. Comme il faut pointer vers un script CGI, nous créons un répertoire **/var/www/test_emu_cgi**, et utilisons la configuration Apache suivante, qui indique de considérer tous les fichiers **.py** comme des scripts CGI exécutés par **mod_python** :

```
<Directory /var/www/test_emu_cgi>
     AddHandler mod_python .py
     PythonHandler mod_python.cgihandler
</Directory>
```

Vous pouvez maintenant reprendre le script donné dans le paragraphe sur le CGI, l'enregistrer dans **/var/www/test_emu_cgi/hello_cgi.py** (inutile de le rendre exécutable), puis visualiser le résultat à l'adresse http://localhost/test_emu_cgi/hello_cgi.py. L'émulateur CGI de **mod_python** n'est

pas destiné à être utilisé en production, mais doit plutôt être utilisé comme une étape de migration d'un script CGI vers **mod_python**. Comme l'émulation se fait au niveau de Python, il peut y avoir des différences de fonctionnement avec un vrai environnement CGI.

2.3 Python Server Pages (PSP)

PSP est plus ou moins équivalent à PHP dans son principe : on écrit une page HTML en incorporant du code Python. C'est éventuellement utile pour programmer une petite application web extrêmement rapide sans s'encombrer d'un framework. Voici comment activer PSP : nous créons un répertoire **/var/www/mod_python_psp/**, puis la configuration Apache suivante, qui demande que tous les fichiers **.psp** soient traités par le handler PSP :

```
<Directory /var/www/test_psp>
    AddHandler mod_python  .psp
    PythonHandler mod_python.psp
</Directory>
```

Ensuite, nous pouvons créer un fichier PSP, et mélanger allègrement du HTML et du Python. Les blocs de code doivent être entourés par **<% ... %>**, les expressions par **<%= ... %>**, les directives (comme **include**) par **<%@ ... %>**, et les commentaires par **<%-- ... --%>**. Voici le fichier **/var/www/test_psp/hello.psp**, dont vous pouvez voir le résultat à l'adresse http://localhost/test_psp/hello.psp :

```
<html>
<%
from datetime import datetime
%>
<body>
hello, nous sommes le <%=datetime.now().day %>
</body>
</html>
```

Pour des applications conséquentes, cette pratique doit être évitée, car on retombe dans le plus gros défaut du développement web à l'ancienne : le mélange entre la logique et la présentation.

2.4 Publicateur de mod_python

Le publicateur de **mod_python** ressemble beaucoup plus aux frameworks modernes que PSP. Il est capable d'analyser l'URL et de diriger la requête vers une fonction Python ou n'importe quel objet *callable* en lui transmettant les données des formulaires. On peut donc écrire un module Python contenant deux fonctions, par exemple **hello()** et **goodbye()**, et accéder à ces fonctions en mettant leur nom dans l'URL. Tout d'abord, voici la configuration Apache :

```
<Directory /var/www/test_publisher>
    SetHandler mod_python
    PythonHandler mod_python.publisher
</Directory>
```

Ensuite, il suffit de créer le fichier **/var/www/test_publisher/sois_poli.py** :

```
msg = '<html><body>%s</body></html>'
def hello(qui='Monsieur'):
    return msg % ('bonjour ' + qui)
def goodbye():
    return msg % 'au revoir'
```

Le publicateur donne un accès direct à la variable **msg** ou à la fonction **hello()** (ou **goodbye**) via les URL http://localhost/test_publisher/sois_poli.py/msg et http://localhost/test_publisher/sois_poli.py/hello. La fonction accepte un paramètre optionnel et on peut transmettre une variable par **GET** en ajoutant **?qui=Christophe** à l'URL.

3 FastCGI

Le Python, comme le Java, s'utilise plus volontiers dans un processus séparé d'Apache. FastCGI est un protocole créé pour offrir un mécanisme similaire au CGI, mais infiniment plus rapide. Son principe est le suivant : le serveur web dispose d'un module FastCGI (**mod_fcgid** ou **mod_fastcgi**) qui lance l'application FastCGI dans un nouveau processus, puis ouvre une connexion sur ce processus. Il envoie l'environnement CGI et d'autres données à l'application au travers de la connexion, que l'application réutilise pour renvoyer sa réponse et son contenu. Le processus n'est pas arrêté, et peut servir pour une autre requête. L'avantage est que l'application peut se situer sur une machine distincte du serveur, ce qui permet de créer des architectures distribuées. Dans ce cas, une connexion TCP est utilisée. Lorsque l'on entre dans ce genre de procédés, il faut faire attention au vocabulaire : le navigateur est un client pour Apache, mais Apache est un client pour l'application : Apache se connecte à l'application, qui se comporte comme un serveur en acceptant ses requêtes.

Faire un Hello World en FastCGI demande plus de code qu'en CGI, car il faut gérer la connexion, effectuer une boucle pour pouvoir traiter plusieurs requêtes et gérer correctement certains signaux provenant du serveur web. Il est donc préférable d'utiliser une bibliothèque offrant ces fonctionnalités, comme celle fournie avec le SDK FastCGI d'Open Market, écrite en C, mais pour laquelle il existe des emballages en Python. L'exemple suivant est basé sur le *wrapper* **fcgiapp** créé il y a dix ans par Digital Creations, l'ancêtre de Zope Corp. :

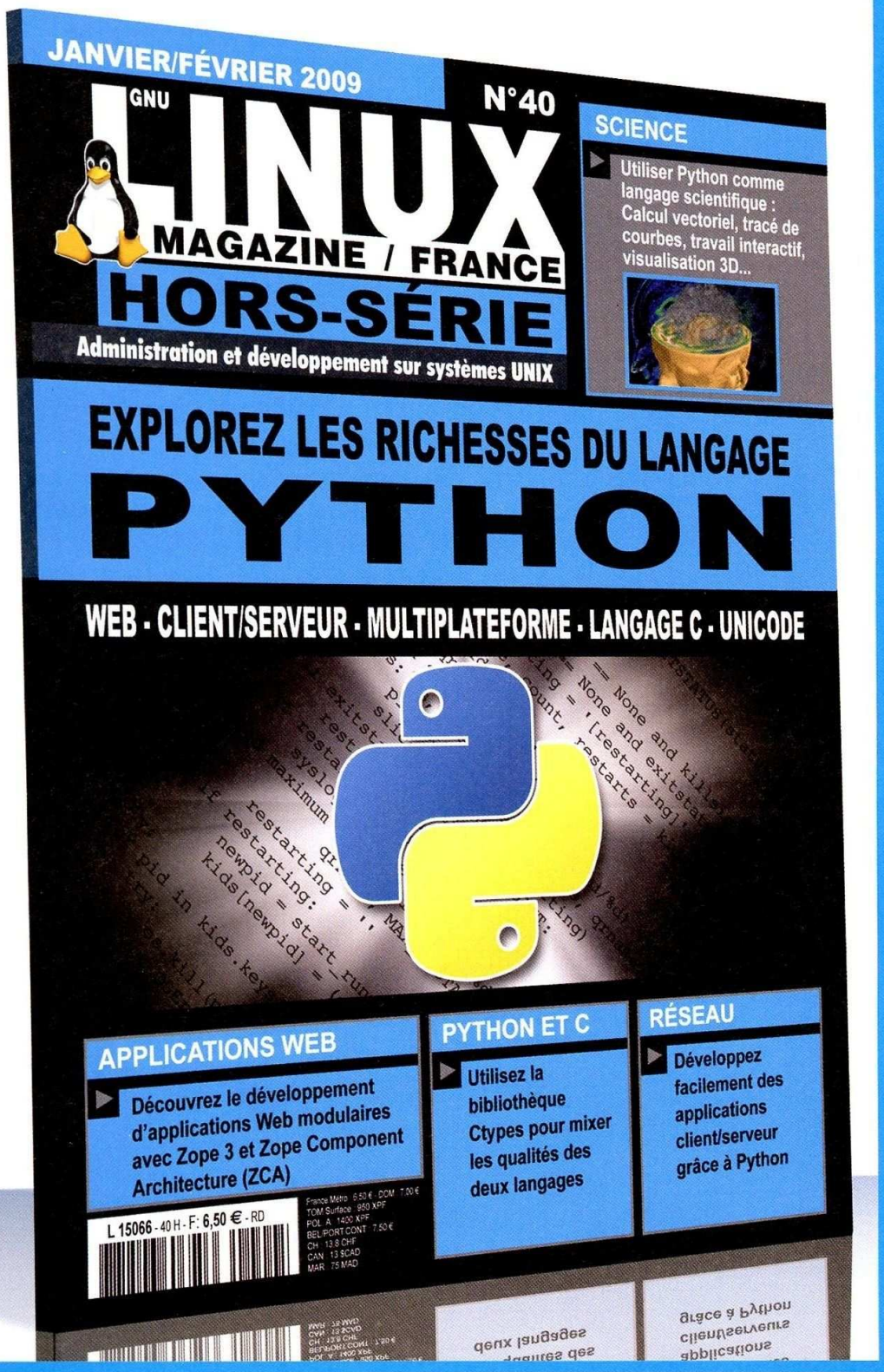
Commencez par installer **mod_fcgid** (le côté Apache) et **fcgiapp** (le côté application) :

```
# aptitude install libapache2-mod-fcgid
# a2enmod fcgid
# /etc/init.d/apache2 restart
# easy_install fcgiapp
```

Pour une connexion locale, il n'y a rien à ajouter côté Apache, la configuration par défaut du module est suffisante (**AddHandler fcgid-script .fcgi**). Il suffit de nommer le script avec une extension **.fcgi**. Pour une connexion TCP, il faudrait ajouter la directive **FastCgiExternalServer** pour indiquer à quelle adresse se situe l'application distante. Créez le fichier **/usr/lib/cgi-bin/hello_fcgi.fcgi** ci-dessous (l'extension doit être **.fcgi**), rendez-le exécutable, puis constatez le résultat à l'adresse http://localhost/cgi-bin/hello_fcgi.fcgi :

ENVIE D'ALLER PLUS LOIN AVEC PYTHON ?

GNU/LINUX MAGAZINE HS 40



AU SOMMAIRE...

Introduction

- Introduction : Python, un monstre de langage
- Nouveautés de Python 2.6
- Nouveautés de Python 3

Éducation

- Apprenez d'abord Python !

Science

- Python comme langage scientifique

Réseau

- Python et le réseau

Code(s)

- Packager et diffuser son application Python
- Trucs et astuces
- Ctypes et Python
- Présentation de la Zope Component Architecture

ENCORE DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 20 MARS 2009

```
#!/usr/bin/python
import fcgiapp
while True:
    input, output, err, environ = fcgiapp.Accept()
    output.write('Content-Type: text/html\n')
    output.write('Status: 200 OK\n')
    output.write('\n')
    output.write('<html><body>hello</body></html>\n')
```

À la différence du CGI, il faut écrire dans le descripteur de fichier **output** au lieu de la sortie standard, et une boucle est présente pour traiter autant de requêtes qu'on veut dans le même processus. Pour faire fonctionner cet exemple, vous devrez définir la variable d'environnement **PYTHON_EGG_CACHE**, car **fcgiapp** a été installé comme un **egg**. Le plus simple est de rajouter ceci au début du script :

```
import os
os.environ['PYTHON_EGG_CACHE']="/tmp"
```

Une autre implémentation FastCGI est disponible dans le paquet **flup** et possède l'avantage de pouvoir servir une application WSGI (voir plus loin pour savoir ce qu'est WSGI). Installez le paquet Debian **python-flup**, puis créez le fichier http://localhost/cgi-bin/hello_fcgi2.fcgi :

```
#!/usr/bin/env python
from flup.server.fcgi import WSGIServer
def app(environ, start_response):
   start_response('200 OK', [('Content-Type', 'text/html')])
   yield '<html><body>hello</body></html>'
WSGIServer(app).run()
```

méthode	nb de requêtes/sec
mod_cgid	40 req/s
mod_fcgid + **fcgiapp**	3300 req/s
mod_fcgid + **flup**	1000 req/s
mod_scgi + **scgi**	1750 req/s
mod_scgi + **flup**	1200 req/s
mod_python (minimaliste)	2100 req/s
mod_python (émulateur CGI)	800 req/s
mod_python (PSP)	1900 req/s
mod_python (publisher)	1400 req/s
WSGI : **wsgiref.simple_server**	900 req/s
WSGI : **mod_cgid** + **wsgiref**	35 req/s
WSGI : **mod_fcgid** + **flup**	1000 req/s
WSGI : **Cherrypy**	2200 req/s
WSGI : **Twisted**	650 req/s
WSGI : **Paste**	1400 req/s
WSGI : **mod_wsgi** (*embedded*)	3050 req/s
WSGI : **mod_wsgi** (*daemon*)	2100 req/s

Benchmarks des exemples utilisés

(test effectué avec Apache Bench (**ab -n 5000 -c 100**), Python 2.5.2, Debian Lenny x86 sur machine virtuelle kvm monoproc sur Core2Duo 2 GHz)

4 SCGI

SCGI est un protocole similaire à FastCGI, mais plus simple. Vous pouvez l'essayer en installant **mod_scgi** pour Apache, puis **python-scgi** pour écrire votre application SCGI en Python :

```
# aptitude install libapache2-mod-scgi python-scgi
# a2enmod scgi
# /etc/init.d/apache2 restart
```

Comme pour le FastCGI, l'application est un serveur auquel Apache se connecte. Dans l'exemple ci-dessous, on utilise une connexion TCP. Il faut indiquer à Apache à quelle adresse se situe le serveur SCGI pour qu'il puisse s'y connecter. Ajoutez la ligne suivante dans sa configuration :

```
SCGIMount /test_scgi/ 127.0.0.1:4000
```

Ensuite, vous pouvez créer votre application dans un fichier quelconque (pas dans **cgi-bin**), avec le nom que vous voulez, par exemple **~/mon_appli_scgi.py** :

```
#! /usr/bin/python
import scgi
import scgi.scgi_server
```

```
class HelloHandler(scgi.scgi_server.SCGIHandler):
    def produce(self, env, bodysize, input, output):
        output.write('Content-Type: text/html\n')
        output.write('\n')
        output.write('<html><body>hello</body></html>')

server = scgi.scgi_server.SCGIServer(
    handler_class=HelloHandler,
    port=4000)

server.serve()
```

Puis, démarrez votre application avec :

```
$ python ~/mon_appli_scgi.py
```

Vous pouvez alors admirer le résultat à l'adresse http://localhost/test_scgi/.

Le paquet **flup** contient également une implémentation SCGI.

Si vous voulez l'essayer, reprenez exactement le code du helloworld FastCGI en **flup**, et remplacez simplement **flup.server.fcgi** par **flup.server.scgi**.

5 L'arme secrète de Python pour le web : WSGI

Le paysage web en Python s'est progressivement ouvert et diversifié après la publication du protocole WSGI (prononcez « whisky » !), sous la forme de la PEP 333 (*Python Enhancement Proposal*) en 2003. Le but de cette PEP est de proposer une interface commune pour tous les serveurs et toutes les applications web en Python,

de façon à pouvoir les brancher librement les uns aux autres. La force de WSGI est non seulement d'être minimaliste et simple à implémenter, ce qui explique sa forte adoption, mais aussi de définir les deux côtés de la chaîne : le côté serveur, et le côté application. Ayant défini ces deux extrémités, il devient possible de créer des composants qui implémentent les deux, et que l'on place entre l'application et le serveur : ce sont les middlewares. Aujourd'hui, la majorité des serveurs et des frameworks web sont compatibles WSGI, et ce protocole peut être utilisé partout, y compris au-dessus de FastCGI, SCGI ou **mod_python**. De nombreux middlewares ont vu le jour et peuvent être branchés très simplement sur n'importe quelle application ou framework. Ils offrent tous les types de services, comme l'authentification, la transmission de sessions, la compression des pages à la volée, l'habillage HTML ou le routage d'URL.

WSGI et les frameworks web

On peut définir un framework comme un ensemble cohérent de méthodes et de briques de programmation conçues pour fonctionner entre elles. Django et Zope en sont deux représentants caractéristiques : ils proposent (et souvent imposent) leurs propres composants et leur propre vision : on développe « en Django » ou « avec Zope » et, avant l'arrivée de WSGI, il fallait obligatoirement faire un choix. WSGI a été initialement présenté comme un tueur de frameworks, car la plupart des fonctionnalités offertes par ces derniers peuvent être prises en charge par des middlewares réutilisables. La notion d'ensemble cohérent est donc devenue plus floue, et les frameworks ont perdu leurs frontières. Mais, au lieu de mourir, ils se sont simplement ouvert à WSGI, et se sont chacun enrichis du travail des autres. Pylons a été l'un des premiers à profiter de cette ouverture et reste en tête de file du monde WSGI.

5.1 Une application WSGI

Côté application, on peut résumer WSGI en trois lignes de code. Voici l'application qui dit « hello » :

```
def application(environ, start_response):
    start_response('200 OK',[('Content-type','text/html')])
    return ['<html><body>hello</body></html>']
```

L'application n'est rien d'autre qu'une fonction ou un objet callable. Le serveur WSGI appelle cette fonction en lui fournissant deux choses :

- un **dict** contenant l'environnement, c'est-à-dire le contenu de la requête et d'autres informations ;
- une fonction de retour, que l'application doit appeler pour transmettre le statut et les en-têtes de réponse.

Après avoir appelé la fonction de retour (**start_response**), l'application doit renvoyer le contenu de sa réponse sous la forme d'un objet *iterable*, ici une liste (ce qui explique l'usage courant de **yield** à cet endroit).

5.2 Des serveurs WSGI

Notre application minuscule peut maintenant être hébergée par n'importe quel serveur compatible WSGI. Il vous suffit d'enregistrer les trois lignes correspondant à l'application dans un fichier (par exemple **mon_appli.py**), puis de rajouter à la fin les quelques lignes qui permettent de la servir, dont voici quelques exemples :

Avec **wsgiref.simple_server** (une implémentation de référence intégrée dans Python) :

```
from wsgiref.simple_server import make_server
server = make_server('localhost', 8080, application)
server.serve_forever()
```

Avec Paste 1.7 :

```
from paste.httpserver import serve
serve(application, 'localhost', 8080)
```

Avec CherryPy 3.1 :

```
from cherrypy import wsgiserver
server = wsgiserver.CherryPyWSGIServer(('localhost', 8080), application)
server.start()
Avec Twisted 8.1 :
from twisted.internet import reactor
from twisted.web2 import wsgi, server, channel
app = wsgi.WSGIResource(application)
site = server.Site(app)
reactor.listenTCP(8080, channel.HTTPFactory(site))
reactor.run()
```

Les quatre exemples précédents fonctionnent entièrement sans Apache. Vous devez juste lancer **python mon_appli.py**, puis vous connecter avec votre navigateur à l'adresse http://localhost:8080. Ces serveurs sont très souvent utilisés en plaçant quand même un Apache en frontal, en mode Reverse Proxy ou URL Rewrite.

Il est également possible de servir notre application directement avec Apache, avec **mod_cgi, mod_fcgid** ou **mod_python** grâce à des adaptateurs :

En CGI :

```
from wsgiref.handlers import CGIHandler
CGIHandler().run(application)
```

En FastCGI :

```
from flup.server.fcgi import WSGIServer
WSGIServer(application).run()
```

5.3 Un module Apache dédié au WSGI !

mod_wsgi, qui est le plus récent et le plus prometteur des modules Apache pour Python, permet de servir nativement une application WSGI. Ses performances sont excellentes. Il est facile à configurer et propose deux modes de fonctionnement : un mode embarqué sur le principe de **mod_python**, et un mode à processus séparé comme en FastCGI. Sur une Debian, l'installation se fait de la façon suivante :

```
# aptitude install libapache2-mod-wsgi
# a2enmod wsgi
# /etc/init.d/apache2 restart
```

Ensuite, il suffit d'une ligne dans un virtualhost d'Apache :

```
WSGIScriptAlias /mon_appli /var/www/mon_appli.wsgi
```

Et notre application de trois lignes, enregistrée dans **/var/www/mon_appli.wsgi**, sera servie directement par Apache à l'adresse http://localhost/mon_appli. Le mode par défaut est le mode embarqué, mais vous pouvez passer au mode « daemon » en ajoutant les directives **WSGIDaemonProcess** et **WSGIProcessGroup**.

5.4 Un middleware WSGI

Nous avons évoqué les middlewares à plusieurs reprises dans cet article. Si le terme paraît pompeux, l'implémentation est ridiculement simple. Voici un exemple du middleware le plus petit possible, pour bien comprendre le principe. Celui-ci ne fait strictement rien :

```
class Middleware1(object):

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        return self.app(environ, start_response)
```

Un middleware est une classe qu'on initialise en lui transmettant l'application dans le constructeur (**__init__**). Il est lui-même callable, c'est-à-dire qu'il se comporte comme une application WSGI, et renvoie un résultat : ici, celui de l'application. On peut donc créer l'ensemble application + middleware comme ceci :

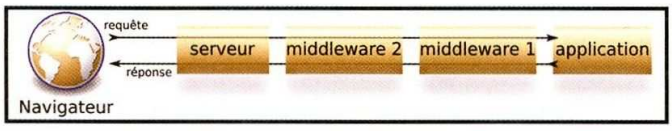
```
appli_modifiee = Middleware1(application)
```

Si on écrit un deuxième middleware, on peut enchaîner les deux en écrivant :

```
appli_modifiee_deux_fois = Middleware2(Middleware1(application))
```

Cette nouvelle application modifiée peut alors être servie avec n'importe quel serveur WSGI, comme on l'a vu un peu plus haut. Dans la pratique, on n'assemble pas les applications, les middlewares et les serveurs à la main de cette façon, mais en profitant des facilités de configuration d'outils comme Paste, dans un fichier de type **.ini**. (cf. la configuration de Pylons).

La représentation graphique qui vient le plus naturellement à l'esprit est celle issue de l'analogie avec la tuyauterie : les middlewares sont des tuyaux qu'on aligne les uns aux autres. Le serveur et l'application sont chacun à une extrémité :



En réalité l'analogie la plus correcte vis-à-vis de l'implémentation est celle de l'imbrication, puisque chaque middleware englobe le suivant, et le premier d'entre eux est le seul visible par le serveur :

Le middleware extérieur est le premier à intercepter la requête venant du serveur. Il peut donc en faire ce qu'il veut : soit la traiter lui-même, soit la transmettre à l'application (ou au middleware suivant). Il se situe donc en amont de l'application, et peut par exemple transformer la requête (URL *rewriting*), répondre à la place de l'application (système de cache) ou diriger vers une application différente (**UrlMap**). S'il décide de transmettre la requête à l'application, c'est aussi lui qui récupère sa réponse et il peut en faire ce qu'il veut : il se situe donc également en aval de l'application, peut transformer sa réponse et faire par exemple de l'habillage en HTML (voir le paquet **deliverance**). Enfin, le middleware peut aussi ajouter ou modifier des informations dans l'environnement (le **dict** environ, le premier paramètre de la fonction WSGI), ce qui peut être utilisé pour de l'authentification (voir les paquets **repoze.who** et **repoze.what**)

Quand on observe l'implémentation minimale d'un middleware, on note une analogie avec le *design-pattern* d'adaptation tel qu'utilisé dans la *Component Architecture* de Zope 3. Le principe est proche. Il s'agit d'enrober un composant avec un autre pour le modifier sans toucher à son implémentation. Dans le cas du middleware, le but est de modifier le comportement global d'une application ; dans le cas de l'adaptateur, il s'agit d'implémenter une nouvelle interface à partir d'une première. Mais, dans les deux cas, on se situe bien dans le cadre de la programmation par composants : les middlewares sont des composants de haut-niveau, les adaptateurs sont des composants de bas niveau. Les deux approches sont complémentaires.

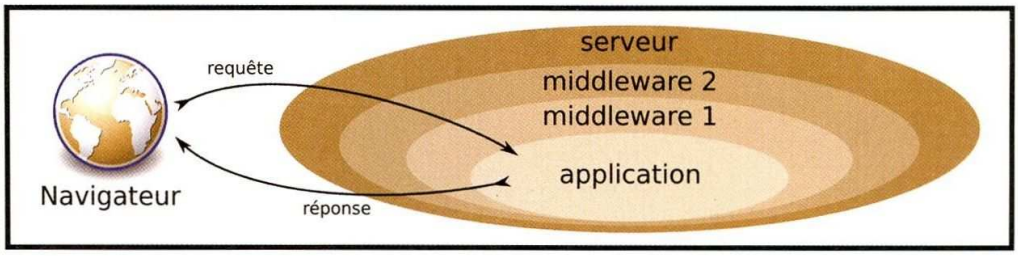
Après lecture de cet article, vous devriez être en mesure de créer de petites applications en Python et expérimenter les principes de base. L'article suivant vous permet d'aller plus loin grâce à l'utilisation de frameworks web.

Références

- WSGI : http://wsgi.org
- Paste : http://pythonpaste.org
- mod_wsgi : http://www.modwsgi.org
- FastCGI : http://www.fastcgi.com http://fastcgi.coremail.cn
- SCGI : http://python.ca/scgi/
- mod_python : http://www.modpython.org
- CGI : http://docs.python.org/library/cgi.html

Auteur : Christophe Combelles

http://ccomb.gorfou.fr – ingénieur freelance
conseils et relectures : Philippe Biondi



BESOIN DE PROTÉGER VOTRE RÉSEAU ?

GNU/LINUX MAGAZINE HS 41

* SOUS RÉSERVE DE TOUTES MODIFICATIONS

AU SOMMAIRE...

- Reportage à NFWS : Convention 2008 des développeurs NetFilter
- Interview : Patrick McHardy, chef du projet NetFilter
- Interview : Pablo Neira, initiateur des conntrack-tools et membre de la coreteam NetFilter
- Interview : David S. Miller, mainteneur de la pile TCP/IP du noyau Linux
- Introduction au filtrage réseau avec NetFilter/IPTables
- Gardez une trace de tout ce qui passe avec Ulogd2, journalisation avancée avec NetFilter
- Passez les paquets à filtrer à une application avec NFQueue
- Snort inline : La détection d'intrusion, facile à configurer !
- Configurez NetFilter pour le filtrage du protocole IPv6
- Amon, le pare-feu de l'Éducation nationale
- Chez les amis d'en face : Découvrir le Packet Filter d'OpenBSD/FreeBSD/NetBSD

DISPONIBLE DÈS LE 20 MARS 2009 CHEZ VOTRE MARCHAND DE JOURNAUX

Trois frameworks web en Python :

Plusieurs facteurs décisifs ont contribué à faire aujourd'hui de Python une plateforme idéale et rassurante pour le web. Tour d'horizon des frameworks Web de Python : Pylons, Django et Zope 3.

Auteurs

- Christophe Combelles
- David Larlet

1 Pourquoi un framework ?

Toutes les techniques présentées dans l'article précédent, depuis le CGI jusqu'au WSGI, sont des techniques de bas niveau pour le traitement des requêtes et l'exécution du code Python. Bien qu'elles soient plus variées, elles laissent le développeur face aux mêmes problématiques que ce qu'offre PHP seul : un langage de programmation ne suffit pas. Lors du développement d'une application web, on constate que les mêmes besoins reviennent sans cesse. Il faut se connecter à une base de données, créer, lire, modifier ou effacer des enregistrements, stocker des informations dans une session, convertir et valider les données des formulaires, récupérer des informations dans la requête, interpréter l'URL... la liste est longue ; ceux qui se sont amusés à développer des applications à la main (sans *framework*) en passant des mois à réinventer la roue pourront confirmer. L'exercice est non seulement ennuyeux, peu rentable, mais également périlleux pour la sécurité.

Les besoins peuvent se découper au minimum en trois groupes : 1. gérer les données et leur logique métier, 2. gérer l'affichage, 3. gérer la logique applicative. C'est ce découpage qu'on appelle MVC : Modèle, Vue, Contrôleur. Le but est de rendre ces trois parties indépendantes et de pouvoir les développer, les modifier et les tester séparément. Le découpage MVC n'est pas seulement pratique, il est absolument vital pour s'assurer qu'une application sera fiable et maintenable. Il est possible d'effectuer un découpage encore plus fin grâce à la *component architecture* qui permet d'isoler les fonctionnalités de bas niveau. Si celle-ci s'avère inutile pour la plupart des projets petits ou moyens, elle devient indispensable pour gérer la complexité des grosses applications ou des gros CMS.

Le but d'un framework est donc de subvenir au minimum à tous les besoins courants ou bien d'offrir la modularité permettant de déléguer ces besoins à des outils tiers. De ce point de vue, les trois framework présentés dans la suite de ce numéro sont assez différents : Django s'efforce de fournir le maximum de manière cohérente et intégrée, Pylons fournit le strict minimum et délègue le reste à des outils externes, Zope 3 offre une multitude de composants interchangeables. Tous les trois sont déployables avec une ou plusieurs des techniques vues précédemment, en particulier WSGI. Ils intègrent notamment un publicateur qui transforme la requête en objet Python et un système pour analyser l'URL et diriger vers le code voulu. Selon votre propre expérience et votre propre sensibilité, vous serez spontanément attiré par un framework, mais n'occultez jamais complètement les autres, car il y a toujours beaucoup à apprendre.

Pylons django ZOPE 3

2 Bref historique

Comme n'importe quel logiciel, chaque framework web a son propre historique, lié à son équipe, aux besoins initiaux, à son mode de développement, à ses évolutions successives et à sa notoriété. Autant de paramètres qui mènent à des différences parfois radicales, qui ont parfois tendance à se gommer avec le temps, mais qui ne disparaissent jamais complètement. Ce sont ces différences qui donnent leur particularité et leur intérêt à chaque framework.

2.1 Pylons

Pylons est un des frameworks les plus récents, puisqu'il est apparu fin 2005 suite à la convergence de projets similaires, le framework Bricks et le moteur de *templating* Myghty, lui-même inspiré de `HTML::Mason`. Il fait partie de la lignée des héritiers de Ruby On Rails, en suivant scrupuleusement le modèle MVC et en limitant les besoins de configuration et de répétition. Sa particularité est d'avoir immédiatement suivi et utilisé le protocole WSGI pour assembler des composants et isoler des fonctionnalités.

En 2006, la version 0.9 a été l'occasion de casser les liens trop forts avec Myghty en le rendant optionnel, et d'extraire plusieurs fonctionnalités dans des paquets indépendants (`WebOb`, `WebError`, `WebTest`, `Beaker`). L'aboutissement de ce processus engendrera la version 1.0 pour début 2009.

2.2 Django

Django est né de la nécessité de développer rapidement des sites dynamiques.

L'équipe au sein du journal *Lawrence Journal-World* en avait assez de réécrire encore et toujours les mêmes bases pour chaque besoin auquel ils étaient confrontés. En 2003, Adrian Holovaty et Simon Willison ont donc commencé à élaborer un framework facilitant le développement web en Python (à l'époque, peu d'outils existaient dans le domaine). Les principaux objectifs étaient d'avoir une base stable permettant d'écrire ses propres applications et de découpler le travail dédié aux développeurs de celui réalisé par les graphistes et intégrateurs.

Ce framework, qui a pour nom Django, en hommage au célèbre guitariste Django Reinhardt dont les développeurs initiaux sont fans, a été libéré en 2005 et est aujourd'hui développé et maintenu par des centaines de développeurs bénévoles. Dernier jalon dans l'ouverture d'une base de code, la *Django Software Foundation* a été créée en juin 2008.

Django est actuellement en version 1.0.2. Après plusieurs années de maturation, le framework a atteint la version 1.0 en septembre 2008 et a adopté un cycle de *releases* rapide sur environ 6 mois. La version 1.1 est prévue pour mars 2009 et apportera son lot de nouvelles fonctionnalités avec notamment de nouvelles options pour l'administration, le support de l'agrégation au niveau de l'ORM ou des vues génériques basées sur les classes Python.

Django forme un ensemble d'outils cohérents permettant de construire son propre site web. Il est donc prédisposé à la création de sites dynamiques et évolutifs. Le framework se base sur des conventions permettant de développer plus rapidement et de manière cohérente avec l'ensemble de la communauté et/ou des intervenants sur un projet. Un mécanisme d'applications découplées permet une réutilisation importante du code. Vous pouvez bien sûr développer les vôtres, mais il existe déjà des milliers d'applications que vous pourrez réutiliser pour vos propres projets.

2.3 Zope 3

Zope est né vers la fin des années 90, de l'agrégation de plusieurs projets développés par la société Digital Creations devenue Zope Corp quelques années plus tard. « Zope » est un acronyme récursif signifiant « *Zope Object Publishing Environment* » et est publié sous la licence ZPL, compatible GPL. Dès le début, Zope a été très fortement orienté objet : son principe de base consistait déjà en de la publication d'objets, et sa base de données est une base de données objet (la ZODB). Zope a séduit beaucoup de monde au début des années 2000 grâce à son interface permettant de développer des applications et des sites entièrement depuis un navigateur web. Il a engendré de grosses applications de gestion de contenu comme Plone qui reste encore aujourd'hui une référence et continue d'évoluer.

En parallèle de la version 2, Zope a été progressivement réécrit en utilisant une architecture innovante à base de composants. Cette réécriture porte le nom de Zope 3 et n'est pas compatible avec la version 2. Il s'agit d'un nouveau framework, qui conserve tous les points positifs de son prédécesseur (la ZODB, le templating, la notion de publication d'objets), mais corrige ses faiblesses et ses défauts. Par contre, son audience est différente : Zope 3 s'adresse à un public plus technique, a perdu son côté monolithique et intégré, et s'apparente plus à une grosse bibliothèque de composants. Il est aussi plus ouvert et interopérable, et parle couramment le WSGI.

La version actuelle est Zope 3.4, sortie deux ans après la version 3.3 en raison d'un important travail de découpage en *eggs* Python, publiés sur l'index des paquets Python (PyPI). Chaque egg suit aujourd'hui son propre cycle de publication. Ainsi, les évolutions et les expérimentations se font plus rapidement, sans attendre la publication du framework complet.

3 Points forts et points faibles

Le framework idéal est celui qui répond le mieux aux besoins d'un projet particulier. Chacun possède des qualités, des défauts, mais surtout des domaines de prédilection et il peut être intéressant d'en connaître plus d'un pour savoir faire face à différents types de demande. Le gros avantage de Python pour le web est d'offrir un très vaste choix, allant des

technologies pérennes et éprouvées jusqu'aux expérimentations récentes les plus prometteuses. Vous en trouverez toujours une qui vous donnera entière satisfaction. Le choix ne doit pas non plus vous effrayer, car aucun framework Python ne vous mènera à une impasse : la norme WSGI apporte rapidement la convergence et l'interopérabilité entre les frameworks et rend possible le partage et la réutilisation du code. En outre, la croissance continue du langage et de sa communauté, ainsi que le support apporté par certains des plus gros acteurs comme Google ou Sun Microsystems est un gage de confiance. Voici donc les principales forces et faiblesses des frameworks ici présentés.

3.1 Pylons

Le point fort de Pylons est son côté minimaliste qui permet de plonger très rapidement dans le développement MVC et d'en comprendre le fonctionnement et les implications, sans se préoccuper de notions comme l'ORM ou le templating. L'intérêt est de laisser le développeur s'exprimer sans lui imposer trop de concepts, ni d'apprentissage. Quasiment rien n'est imposé, et on peut utiliser le framework comme on veut, avec les outils auxquels on est habitués. L'autre gros intérêt de Pylons est son adhérence stricte à WSGI et son utilisation à tous les niveaux. WSGI est un standard intégré à Python, et c'est lui qui permet à Pylons d'être à la fois tout petit, mais de fournir tous les types de services grâce à des *middlewares* optionnels.

Le côté minimaliste de Pylons constitue aussi son point faible. Le fait d'avoir trop de choix est effrayant pour les débutants. Pourquoi utiliser FormAlchemy plutôt que FormEncode ? SQLAlchemy plutôt que Storm ? Mako plutôt que Genshi ? Choisir en connaissance de cause demande un peu d'expérience. Toutefois, le site web de Pylons donne quelques pointeurs sur les choix conseillés et vous pouvez les suivre les yeux fermés, car ce sont des valeurs sûres.

3.2 Django

La principale force de Django réside dans sa cohérence qui permet une grande rapidité de prise en main. Que ce soit pour de la documentation ou du support, vous n'avez qu'un seul endroit où aller chercher l'information, ce qui fait gagner un temps non négligeable à l'usage.

Malheureusement, sa principale force est aussi sa principale faiblesse, car cette unité se fait aux dépens de la flexibilité du framework. Il reste possible d'utiliser des parties tierces (comme Jinja2 pour les templates ou SQLAlchemy pour les modèles), mais c'est fastidieux. Cela dit, les choix par défaut sont pertinents et devraient parfaitement convenir à votre besoin. Dans le cas contraire, vous aurez normalement les compétences pour intégrer les modules tiers.

L'utilisation d'unicode et la gestion facilitée des traductions fait aussi la différence dans le cadre de projets en plusieurs langues. C'est un point généralement important pour nous, francophones.

Enfin, tout est fait pour vous encourager à effectuer des tests (sous la forme de tests unitaires ou de *doctests*), ce qui est une garantie de la pérennité et de l'agilité de votre application, gages de sa qualité.

3.3 Zope 3

L'intérêt de Zope 3 vient de la maturité de son code et de sa communauté. Il s'agit d'un framework développé depuis 10 ans, qui a été largement utilisé, évalué et critiqué dans sa version 2, puis complètement repensé et réécrit en tenant compte de l'expérience et des erreurs commises. Cette maturité a accompagné la croissance des besoins techniques et conceptuels des membres et des entreprises de la communauté, et des besoins de solutions efficaces pour la réutilisation de code. Elle a pris la forme de la component architecture et de la mise en œuvre de la programmation par contrat. Ces deux principes existent aussi sur d'autres grosses plateformes de développement, mais grâce à la simplicité et à l'élégance du langage Python, ils sont plus faciles à appréhender et permettent d'envisager sereinement des hauts niveaux d'abstraction.

Le point faible de Zope 3 est une conséquence de son découpage en eggs et de sa récente nature de bibliothèque de composants. La documentation est éparpillée dans de très nombreux paquets et n'est parfois constituée que de doctests. Elle trouve quand-même une bonne visibilité au travers de l'*apidoc*, dont vous pouvez consulter une version statique sur le site apidoc.zope.org. Les doctests permettent de garantir que la documentation est à jour, mais les développeurs ne pensent parfois pas à en extraire une documentation de haut niveau ou un tutoriel. Une bonne partie de ces efforts est en réalité déportée vers des projets complémentaires ou parallèles de la communauté Zope, comme le CMS Plone ou les très récents frameworks Grok et **repoze.bfg** qui font tous usage des composants Zope 3 et de la component architecture. Heureusement, une initiative de refonte et de réorganisation du site zope.org est en cours et semble sur le point d'aboutir (voir le site new.zope.org).

4 Tutoriel de démarrage

Un point commun à tous les frameworks est qu'ils fournissent des outils de démarrage rapide d'un projet. Certains outils comme Paste et Buildout finissent même par être utilisés par plusieurs frameworks tant ils sont utiles. Vous trouverez dans cette section un tutoriel pour vous aider à démarrer avec chacun, ainsi qu'un petit exemple d'utilisation. Pour éviter les conflits avec les paquets préinstallés sur votre système, il est conseillé d'utiliser Virtualenv. Le tutoriel sur Pylons l'utilise et vous pouvez vous en inspirer pour faire la même chose avec Django et Zope 3.

4.1 Pylons

Pylons est probablement déjà fourni par votre distribution Linux. Le langage Python, comme Java avec les jar ou Ruby avec les gems, fournit son propre système de paquets, les eggs, avec gestion des dépendances et des métadonnées. Vous n'êtes pas obligé de l'utiliser et vous pouvez installer Pylons en installant le paquet fourni par votre système. (**python-pylons** sous Ubuntu). Mais, nous allons plutôt effectuer l'installation dans un bac à sable grâce à Virtualenv, sans avoir besoin d'être administrateur. Vous pourrez ainsi tester Pylons sans aucun impact sur votre système. Dans un projet plus complet qu'un *HelloWorld*, vous pouvez utiliser également l'outil Buildout, qui automatise l'installation et les dépendances. Ici, nous allons nous limiter à Virtualenv.

La première étape est d'installer Virtualenv. Si vous êtes administrateur, vous pouvez faire au choix **sudo aptitude install python-virtualenv** sur une Debian ou Ubuntu ou **sudo easy_install virtualenv** sur n'importe quel système si vous avez déjà installé **setuptools**. Sinon, il existe un script **virtualenv.py** que vous pouvez télécharger et lancer sans rien installer.

Ensuite, on crée et on active notre bac à sable :

```
$ virtualenv bacasable
$ cd bacasable
$ source bin/activate
```

Votre shell est modifié et possède maintenant un PATH qui pointe en priorité vers le dossier **bin** du bac à sable. Ce dossier **bin** contient des commandes **python** et **easy_install** dont le **PYTHONPATH** pointe en priorité vers le dossier **site-packages** du bac à sable. Vous pouvez donc simultanément profiter de vos bibliothèques système et en installer de nouvelles dans le bac à sable. Si vous voulez être à 100% isolé de vos bibliothèques système, ajoutez l'option **--no-site-packages** à Virtualenv. Nous pouvons maintenant installer Pylons sans être administrateur :

```
$ easy_intall Pylons
```

Cette commande interroge l'index des paquets Python (**pypi.python.org**) pour trouver Pylons, récupère sa dernière version (0.9.7), puis télécharge les dépendances suivantes :

- **Tempita** : un mini-moteur de templating ;
- **WebTest** : un utilitaire de test fonctionnel pour les applications WSGI ;
- **WebError** : des outils et des middlewares de capture et de gestion d'erreurs ;
- **WebOb** : des objets pour gérer la requête et la réponse en environnement WSGI ;
- **Mako** : un moteur de templating très rapide, semblable à PSP (voir **mod_python**) ;
- **Nose** : un outil de détection et de lancement des tests unitaires ;
- **decorator** : un module pour simplifier l'usage des décorateurs Python ;
- **simplejson** : une bibliothèque d'encodage/décodage JSON ;
- **FormEncode** : un outil de validation, génération et conversion de formulaires web ;
- **Paste** : le vrai couteau suisse WSGI (et le plus gros, avec toutes les lames) : contient un serveur web, des middlewares, des outils de test, de validation, de compression, de session, etc. ;
- **PasteScript** : un script de gestion extensible, permettant entre autres de créer des squelettes d'applications ou de lancer le serveur ;
- **PasteDeploy** : l'outil de configuration et d'assemblage des applications, middlewares et serveurs WSGI ;
- **Beaker** : un middleware de mise en cache et de session web ;
- **WebHelpers** : de nombreux assistants dédiés au web ou au codage Python en général ;
- **Routes** : un moteur de routage d'URL vers le code Python.

Après installation, un nouveau script **paster** devrait se trouver dans le dossier **bin**. Nous pouvons l'utiliser pour créer notre projet vide :

```
$ paster create -t pylons monprojet
```

La première chose que fait cet outil est de vous laisser choisir un système de templating ; cela donne un avant-goût du principe adopté par ce framework : vous avez le choix ! Rien ne vous est imposé, vous pouvez utiliser les outils que vous voulez. Le choix par défaut est un système de templating à balises : **Mako**. Puis, vous devez indiquer si vous souhaitez utiliser SQLAlchemy pour définir et stocker vos données (répondez non pour ce tutoriel), et enfin si votre application doit fonctionner sur Google App Engine (voir l'encadré plus bas).

Vous disposez maintenant d'un dossier **monprojet**, contenant un paquet Python **monprojet** et plusieurs sous-paquets, **model**, **controller**, **templates**, etc. qui correspondent aux divers éléments de votre application. Il n'en faut pas plus pour avoir une application web fonctionnelle, que vous pouvez lancer comme ceci :

```
$ cd monprojet
$ paster serve --reload development.ini
```

Consultez l'adresse http://localhost:5000 avec le navigateur de votre choix et admirez le magnifique index par défaut ! Tellement joli que... nous allons le supprimer. (Jetez-y quand-même un œil, il contient des liens vers la documentation et d'autres choses intéressantes) :

```
$ cd monprojet
$ rm public/*
```

Inutile de redémarrer le processus **paster**, l'option **--reload** permet de le faire à votre place dès que vous faites un changement dans votre code. Pour un gabarit de page, le redémarrage est d'ailleurs inutile. Vous pouvez tout de suite écrire le nouveau fichier **public/index.html** ci-dessous et voir son résultat à la même adresse :

```
<html><body>
  <div>
    Ca marche !
  </div>
</body></html>
```

Le dossier **public** abrite les contenus statiques et publics. Tout ce que vous y mettrez sera disponible sans restriction.

4.2 Django

Il est nécessaire d'avoir une version de Django installée sur sa machine pour pouvoir réaliser ce tutoriel qui n'a pas pour but de vous guider dans ces étapes. N'hésitez pas à vous référer à la documentation sur le sujet : http://docs.djangoproject.com/en/dev/topics/install/.

Pour vérifier votre installation, la commande suivante devrait vous afficher la version installée :

```
$ python -c "import django; print django.get_version()"
```

Dans un premier temps, nous allons créer une application toute simple permettant d'afficher le traditionnel message de test.

Pour cela, il faut commencer par créer un projet à l'aide de l'utilitaire en ligne de commande qui devrait normalement se trouver dans vos commandes exécutables, quel que soit votre type d'installation (si ce n'est pas le cas, il est dans **django/bin/**) :

```
$ ./django-admin.py startproject projet_hors_serie
```

Cette commande a dû créer une arborescence de fichiers que nous détaillerons plus tard. Vous pouvez dès à présent vous rendre dans le dossier créé et lancer le serveur de développement avec la commande :

```
$ cd projet_hors_serie
$ python manage.py runserver
```

Rendez-vous à l'adresse http://127.0.0.1:8000/ comme le suggère le message qui vient de s'afficher et vous devriez normalement avoir confirmation du bon fonctionnement de votre projet.

Un projet est découpé en plusieurs applications. Dans notre cas, nous allons créer l'application **hello_world**. Une fois dans le dossier du projet précédemment créé, lancez la commande suivante :

```
$ python manage.py startapp hello_world
```

Normalement, cette deuxième commande a aussi généré une arborescence de fichiers que nous explorerons au fur et à mesure. Pour l'heure, intéressons-nous au fichier **settings.py** du projet. Il va y falloir déclarer que nous utilisons l'application **hello_world** en ajoutant une ligne au paramètre **INSTALLED_APPS** :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites'
)
```

devient :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'hello_world',
)
```

Passons maintenant aux URL du projet avec le fichier **projet_hors_serie/urls.py**. Il y a déjà une ligne commentée qu'il suffit d'activer et d'adapter

```
urlpatterns = patterns('',
    # Example:
    # (r'^projet_hors_serie/', include('projet_hors_serie.foo.urls')),

    # Uncomment this for admin:
#     (r'^admin/', include('django.contrib.admin.urls')),
)
```

qui devient :

```
from hello_world.views import hello

urlpatterns = patterns('',
    # Example:
    (r'^hello_world/$', hello),
)
```

On définit ainsi que l'URL **hello_world** va renvoyer sur la fonction **hello** qui se trouvera dans l'application. Allons tout de suite la créer en éditant **projet_hors_serie/hello_world/views.py** :

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('Hello world!')
```

Et voilà, si vous relancez la commande pour activer le serveur local, vous devriez pouvoir vous rendre à l'adresse http://127.0.0.1:8000/hello_world/ et vérifier que la vue a bien renvoyé le bon message.

Détaillons ensemble ce qu'il vient de se passer. La requête envoyée est arrivée dans les URL où l'expression régulière a évalué la chaîne **hello_world** et a trouvé la vue hello associée. Cette vue retourne ensuite le contenu « Hello world! » dans une réponse.

Mais, la force de Django, ce sont aussi ses templates. Dynamisons la page afin qu'elle utilise aussi les paramètres de requête. Commençons par créer un dossier **templates** dans le dossier du projet et un fichier **hello.html** à l'intérieur de celui-ci. Ce fichier devra contenir :

```
Hello {{ username }}!
```

Il faut maintenant passer le paramètre **username** au template au niveau de la vue **hello** qui devient :

```
from django.views.generic.simple import direct_to_template

def hello(request):
    return direct_to_template(request, 'hello.html',
        extra_context={'username': request.GET.get('username', 'world')})
```

La fonction **direct_to_template** est un raccourci pour renvoyer une réponse via un template avec des variables contenues dans un contexte.

Il ne reste plus qu'à déclarer dans les **settings.py** où trouver le template en question :

```
TEMPLATE_DIRS = (
    'templates',
)
```

Vous pouvez maintenant rafraîchir la page précédente et vous devriez avoir un affichage similaire. Par contre, si vous ajoutez un paramètre **username** à la requête, par exemple **?username=david**, le template devrait utiliser la variable passée dans la vue.

C'est suffisant pour démarrer. Les prochaines parties devraient vous permettre de mieux comprendre les mécanismes internes et d'apprendre à utiliser les modèles permettant de stocker vos données.

4.3 Zope 3

Le mode d'installation et d'utilisation de Zope a changé : il ne s'agit plus de télécharger et décompresser une grosse archive contenant tout, car Zope est réparti dans de nombreux paquets. Par conséquent, il est fourni avec un outil permettant de créer un squelette minimal et de choisir les dépendances dont on a besoin. Zope 2 était une grosse application monolithique et votre application était juste un « produit », un *plugin* de Zope, en quelque sorte. La tendance s'est inversée : Zope s'est transformé en bibliothèque de composants et est une simple dépendance de votre application. C'est votre application qui est au centre et plus le framework.

La première étape consiste donc à installer l'outil de démarrage rapide Zopeproject et l'outil de construction et de déploiement Buildout. Vous pouvez éventuellement créer un Virtualenv pour être vraiment isolé (voir le tutoriel Pylons) :

```
$ sudo easy_install-2.5 zopeproject zc.buildout
```

Vous pouvez maintenant lancer Zopeproject en donnant le nom de votre projet. Zopeproject n'est en réalité qu'un enrobage autour de PasteScript. Donc, la procédure est très proche de celle de Pylons. Cette commande vous demande de choisir un *login* et mot de passe, ainsi que le chemin du dossier où seront stockés tous les eggs téléchargés, puis elle crée un dossier **monprojet** avec le minimum indispensable.

```
$ zopeproject --no-buildout monprojet
$ cd monprojet
```

Zopeproject a terminé son travail et est maintenant inutile pour votre projet : son seul rôle était de créer un squelette d'application, le relais est passé à l'outil de construction Buildout, qui est une sorte de **make** de haut niveau.

Le projet créé par la version actuelle (0.4.1) de Zopeproject pointe par défaut vers les dernières versions de tous les paquets Zope, ce qu'on peut comparer à une version en développement comme Debian Unstable. Ceci n'est absolument pas conseillé. La version stable à utiliser correspond à une liste de versions de paquets qui sont tous compatibles et testés dans leur ensemble, et qu'on peut consulter à l'adresse http://download.zope.org/zope3.4/3.4.0/versions.cfg. Cet ensemble de versions constitue le KGS (*Known Good Set*) et sert à définir la version majeure actuelle 3.4 de Zope. Pour pointer vers cette version stable, il faut l'indiquer dans le fichier de configuration **buildout.cfg**.

Ouvrez le fichier **buildout.cfg**, et remplacez la ligne

```
find-links = http://download.zope.org/distribution
```

par :

```
extends = http://download.zope.org/zope3.4/3.4.0/versions.cfg
versions = versions
```

Pour éviter un accès réseau à chaque lancement de Buildout, vous pouvez également télécharger **versions.cfg** et écrire **extends = versions.cfg**.

Parmi les fichiers créés, on trouve :

- **setup.py** est le fichier classique (**setuptools/distutils**) de configuration/installation de votre application en tant que paquet Python. C'est dans ce fichier que doivent apparaître le nom et la version de votre application, les dépendances et les paquets tiers dont vous aurez besoin. Vous pouvez le modifier maintenant (ou plus tard) pour renseigner ces informations.
- **buildout.cfg** décrit l'environnement où va fonctionner votre application. Un Buildout est un environnement isolé, contenu entièrement dans votre dossier **monprojet**. Pour plus d'informations sur Buildout, consultez la documentation à l'adresse http://buildout.zope.org.
- **debug.ini** et **deploy.ini** sont deux fichiers de configuration au choix pour le serveur HTTP de Paste, qui sert l'application en mode WSGI.
- **zope.conf** est le fichier de configuration de Zope lui-même, définissant l'emplacement de la base de données objet et du fichier journal.
- **zdaemon.conf** configure le *daemon* permettant de lancer et surveiller le processus de l'application et de le redémarrer en cas de besoin.

- **var** contient la base de données, **log** les fichiers de log et **src** le code source de l'application.
- **site.zcml** est le fichier de configuration racine des composants Python de votre application. Son rôle principal est de reconstruire en mémoire le registre des composants au démarrage de l'application. Il s'agit du tout premier fichier ZCML lu, tous les autres fichiers ZCML sont inclus depuis cet endroit.

C'est le moment de lancer la construction de votre application :

```
$ buildout bootstrap
$ bin/buildout
```

Le *bootstrap* consiste à installer l'outil Buildout lui-même. Vous obtenez ainsi un script **buildout** dans le dossier **bin**, qu'il suffit de lancer. **buildout** lit son fichier de configuration **buildout.cfg** et construit l'environnement automatiquement en fonction des directives. Si vous l'exécutez pour la première fois, la procédure peut prendre un peu de temps, car les paquets sont téléchargés depuis l'index des paquets Python (http://pypi.python.org/pypi), éventuellement compilés, puis installés dans le Buildout. À la fin de la construction, votre dossier **~/.buildout/eggs** devrait avoir grossi et vous devriez obtenir un script de lancement **bin/monprojet-ctl**. Si vous modifiez le fichier **setup.py** ou **buildout.cfg**, vous devrez relancer la commande **bin/buildout** afin de reporter les modifications dans votre environnement.

Vous pouvez alors lancer votre application avec le script **monprojet-ctl** et sa commande **fg** (**foreground**) ou **start** (**background**) :

```
$ bin/monprojet-ctl fg
```

ou bien lancer Paste directement en choisissant sa configuration (**debug.ini** ou **deploy.ini**) :

```
$ bin/paster serve debug.ini
```

De cette façon, vous obtenez un mode *debug* offrant un middleware emprunté à Pylons qui affiche les exceptions dans le navigateur et qui permet d'inspecter les variables et de lancer des instructions Python à n'importe quel niveau de la pile d'appel. Extrêmement utile !

Vous pouvez ouvrir maintenant votre navigateur à l'adresse http://127.0.0.1:8080, vous connecter avec le login que vous avez choisi, et observer l'interface de gestion de Zope, qui permet de naviguer dans la base de données objets et d'obtenir toutes sortes d'informations sur le serveur et les objets eux-mêmes. Après authentification, vous pouvez même ajouter des objets dans la base, par exemple un Dossier (voir la capture d'écran dans l'article « gestion et administration »).

L'autre adresse indispensable est celle de la documentation intégrée. Elle permet de parcourir toute la hiérarchie des classes, de connaître les classes de base, les interfaces implémentées, et de lire la documentation et les doctests de tous les paquets : http://127.0.0.1:8080/++apidoc++/ (ne visitez pas la doc en mode debug sans être d'abord authentifié, sinon vous ne serez pas redirigé vers le formulaire de connexion et verrez apparaître l'erreur d'authentification).

Commençons par afficher une simple page HTML statique. Dans le répertoire **src/monprojet/**, créez un fichier **bonjour.pt** avec le contenu suivant :

```
<html><body>
  <div>
    Ca marche !
  </div>
</body></html>
```

Conventions ou configuration ?

Les déclarations ZCML sont systématiques dans Zope, car ce n'est pas un framework basé sur des conventions (au contraire de Pylons ou Django), mais sur de la configuration : tout est explicite, tout doit être écrit. Si les déclarations XML vous effraient, mais que vous êtes quand même intéressé par les concepts de Zope (base de données objet, architecture de composants), vous devriez probablement essayer Grok, qui est un emballage autour de Zope 3 le rendant plus accessible aux débutants, basé sur des conventions et sans déclaration XML.

Cette page ne peut pas être affichée directement. Il faut d'abord la déclarer au serveur d'application. Ajoutez dans le fichier **src/monprojet/configure.zcml** les lignes suivantes, avant le **</configure>** final :

```
<browser:page name="bonjour.html"
              for="*"
              template="bonjour.pt"
              permission="zope.Public" />
```

Cette déclaration signifie : je veux une page **bonjour.html**, disponible pour tous les types de contenu, dont le gabarit est le fichier **bonjour.pt**, et visible par tout le monde.

Il faut également ajouter la déclaration du **namespace** au début du fichier, en ajoutant **xmlns:browser="http://namespaces.zope.org/browser"** à l'intérieur du tag **<configure ...>** :

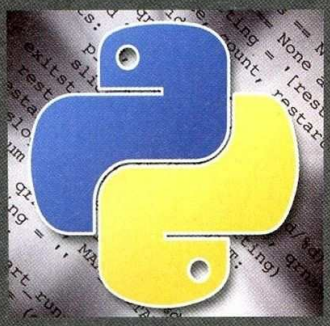
```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:browser="http://namespaces.zope.org/browser"
           i18n_domain="monprojet">
```

Redémarrez l'application après avoir sauvé le fichier et vous pourrez constater que votre page s'affiche à l'adresse http://127.0.0.1:8080/bonjour.html. Le redémarrage est nécessaire si vous modifiez un fichier Python ou ZCML. Pour un gabarit HTML, c'est inutile.

Auteurs : Christophe Combelles, David Larlet

Zope3, Pylons, introductions : Christophe Combelles
Django : David Larlet
Conseils et relectures : Gael Pasgrimaud, Philippe Biondi et Thierry Florac

Trois frameworks web en Python : comparatif technique



Auteurs

- Christophe Combelles
- David Larlet

L'article précédent vous a proposé un survol des trois frameworks. Nous allons maintenant approfondir un peu en détaillant le découpage MVC, puis nous terminerons par un aperçu de leur modèle de sécurité et des interfaces d'administration offertes.

1 Modèles de données et stockage

Cette section concerne la partie « M » du MVC, c'est-à-dire la façon dont sont définies les données et la façon dont elles sont stockées et interrogées. Cette couche est censée abriter la plus grosse partie de la logique métier de votre application. Python étant un langage encourageant la programmation objet, la définition des données se fait à l'aide de classes Python. Si les données sont stockées dans une base de données relationnelles comme SQLite, PostgreSQL ou MySQL, il faut faire appel à un outil qui fait le lien entre les objets Python et les tables de la base de données, c'est-à-dire un ORM (*Object Relational Mapper*). L'ORM se charge de fournir une API objet vous permettant de lire et d'écrire dans la base sans jamais toucher une ligne de SQL. Si les données sont stockées dans une base de données objet comme la ZODB ou Durus, les objets Python sont accessibles directement et l'ORM n'est pas nécessaire.

1.1 Pylons : une alternative proposée par Google

Google a ouvert l'été dernier un service de *cloud computing* appelé *Google App Engine*, pour héberger vos applications en profitant de leur infrastructure. Le principe est de permettre à votre application de croître autant qu'elle veut, aussi bien en puissance de calcul qu'en espace de stockage. Le langage utilisé est une version légèrement restreinte de Python et les données sont stockées dans une base de données qui n'est ni vraiment relationnelle, ni objet, mais qui est une base distribuée supportant de très gros volumes (plusieurs petaoctets) et utilisant un langage de requête spécifique (GQL). Il faut noter que plusieurs tentatives sont en cours pour faire fonctionner les *frameworks* Python sur l'infrastructure de Google App Engine.

Pylons laisse le choix du moyen de persistance au concepteur de l'application. La seule chose fournie est un paquet **model** qui ne contient par défaut rigoureusement rien. Vous pouvez donc utiliser aussi bien des fichiers texte au format ini, que le module **pickle** de Python, une base de données objet comme la ZODB ou n'importe quelle base de données relationnelle, grâce à un ORM comme SQLObject ou SQLAlchemy. SQLAlchemy est d'ailleurs plus qu'un ORM, c'est une couche d'abstraction générique pour les bases de données relationnelles. C'est la méthode conseillée pour stocker vos données dans une base SQL avec Pylons. Lorsque vous créez un nouveau projet avec Paste, le choix d'utiliser SQLAlchemy vous est proposé et le squelette de votre application va refléter ce choix : tout est prêt pour commencer à créer vos modèles et vos tables.

SQLAlchemy peut être utilisé de trois façons : avec le *SQL Expression Language*, avec le *Mapper objet* ou en mode déclaratif entièrement objet.

Le SQL Expression Language permet de dialoguer avec la base de données directement en Python, grâce à une API objet très proche du SQL : les tables, les colonnes, les connexions, instructions SQL sont tous des objets, et peuvent être combinés pour attaquer de manière identique n'importe quelle base de données, SQLite, PostgreSql, MySql, Oracle, MS SQL, etc.

Voici un très court exemple pour stocker la liste de vos peluches (vous allez découvrir Alain le python). Il faut commencer par définir la table et ses colonnes. L'objet **MetaData** sert à conserver la liste des tables, ainsi que le type de base :

```
from sqlalchemy import create_engine, Table, Column,
Integer, String, MetaData
metadata = MetaData()

t_peluches = Table('peluches', metadata,
    Column('id', Integer, primary_key=True),
    Column('nom', String),
    Column('description', String),
)
```

Puis, on peut créer la base et ses tables, en utilisant ici une base SQLite 3 :

```
engine = create_engine('sqlite:///peluches.db', echo=True)
metadata.create_all(engine)
```

Enfin, on peut créer une instruction :

```
instruction = t_peluches.insert().values(nom=u'Alain',
                                description=u"La mascotte de l'AFPY")
```

Et on peut exécuter cette instruction au travers d'une connexion :

```
connexion = engine.connect()
connexion.execute(instruction)
```

Cette méthode vous permet de travailler au plus bas niveau, sans ORM, tout en étant indépendant du type de base utilisée. Des modules spécialisés vous donnent néanmoins accès aux fonctionnalités spécifiques de chaque base.

Un deuxième moyen d'utiliser SQLAlchemy est de se servir de son Mapper objet. Le principe est de faire correspondre un objet Python à une table ou un ensemble de tables, sans que l'objet ait connaissance des tables et de leur structure. Il suffit de créer une classe représentant l'abstraction dont on a besoin

```
class Peluche(object):
    def __init__(self, nom, desc):
        self.nom, self.description = nom, desc
```

puis d'établir la correspondance entre cette classe et les tables précédemment créées :

```
from sqlalchemy.orm import mapper, sessionmaker
mapper(Peluche, t_peluches)
```

On en profite pour créer la classe qui servira de session de connexion, afin de regrouper éventuellement les requêtes dans une transaction. La session est le seul point d'accès réel à la base de données :

```
Session = sessionmaker(bind=engine)
```

On peut de cette façon travailler uniquement avec l'objet **Peluche**, et non avec les tables. Voici comment effectuer la même insertion :

```
alain = Peluche(u'Alain', u"La mascotte de l'AFPY")

session = Session()
session.add(alain)
session.commit()
```

L'intérêt de cette méthode est que notre classe **Peluche** est définie de manière complètement séparée, sans aucun code lié à la base de données, et peut abriter toutes les méthodes ou la logique métier dont on a besoin. La mise en correspondance avec la base de données se fait à part. Elle permet juste d'ajouter à notre classe des attributs correspondant aux colonnes de la table. Inutile de dire qu'un objet peut être mis en correspondance avec plusieurs tables, de toutes les manières possibles, et peut être impliqué dans tous les types de relations.

La dernière méthode, la méthode déclarative, propose de définir tout en une fois : les tables, la classe métier, et la correspondance. Elle réduit la quantité de code à écrire. Voici un exemple complet de déclaration et d'insertion d'une peluche :

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///peluches.db', echo=True)

Base = declarative_base()

class Peluche(Base):
    __tablename__ = 'peluches'

    def __init__(self, nom, desc):
        self.nom, self.description = nom, desc

    id = Column(Integer, primary_key=True)
    nom = Column(String)
    description = Column(String)

Peluche.metadata.create_all(engine)

alain = Peluche(u'Alain', u"La mascotte de l'AFPY")

Session = sessionmaker(bind=engine)
session = Session()
session.add(alain)
session.commit()
```

Dans cet exemple, on crée d'abord l'**engine**, puis la classe **Peluche**. Cette classe contient déjà l'objet **metadata**, qu'on utilise pour créer la base et les tables. Ensuite, on peut instancier **Alain**, et l'insérer dans la base à l'aide de la session. L'utilisation est donc la même, seule la définition du modèle change. Un produit tiers appelé Elixir sert également à créer des modèles de façon déclarative pour SQLAlchemy et mérite votre attention.

Pour récupérer **alain** depuis notre base de données, il suffit de faire une requête en interrogeant le modèle (la classe), puis en filtrant la requête avec l'attribut nom, puis en récupérant le résultat, censé être unique dans le cas de **one()** :

```
requete = session.query(Peluche)
requete = requete.filter_by(nom='Alain')
alain = requete.one()
```

Si on veut obtenir tous les résultats, on peut récupérer directement une liste des objets grâce à **all()**, ce qu'on peut écrire en une ligne :

```
liste_alain = session.query(Peluche).filter_by(nom='Alain').all()
```

L'objet session gère le lien avec la base, notamment les requêtes et les transactions, et l'objet **query** sert à construire des requêtes très évoluées, avec plus d'une quarantaine de méthodes comme **filter**, **group_by**, **join**, **distinct**, **limit**, etc.

1.2 Django

Django dispose d'un ORM permettant d'utiliser des objets Python pour gérer les données persistantes de vos applications web (stockées dans des bases de données, Django supporte nativement SQLite, MySQL, PostgreSQL et Oracle).

Dans la suite de notre exemple, nous montrerons comment créer un blog basique, celui-ci étant doté de billets (titre et contenu) et de commentaires que l'on va définir sous la forme de classes Python.

Commençons par créer l'application blog dans notre projet initialement créé :

```
$ python manage.py startapp blog
```

Cette nouvelle application devrait contenir un fichier **models.py** qu'il vous faut éditer pour créer les modèles. La définition de la méthode **__unicode__** permet d'avoir un rendu plus lisible, mais elle est facultative :

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()

    def __unicode__(self):
        return u"%s" % self.title
```

La table **blog_post** qui sera créée en conséquence de ce modèle aura trois champs :

- un champ **id** de type **INTEGER** qui est automatiquement ajouté par Django pour constituer la clé primaire de la table ;
- un champ **title** de type **VARCHAR** pour le titre ;
- un champ **content** de type **TEXT** pour le contenu.

Et pour les commentaires ? Nous allons utiliser l'application intégrée dans Django. L'une des forces de ce framework est la réutilisation aisée des applications existantes. Pour cela, ajoutons ces nouvelles applications à nos *settings* :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'hello_world',
    'blog',
    'django.contrib.comments',
)
```

Il est nécessaire de spécifier la base de données utilisée pour stocker les données de nos modèles, ici aussi au niveau des settings. Nous allons utiliser SQLite par souci de simplicité :

```
import os
ROOT_PATH = os.path.dirname(__file__)
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = os.path.join(ROOT_PATH, 'testdb.sqlite')
```

Petite astuce ici. On définit le paramètre **ROOT_PATH** qui correspond au dossier courant afin de pouvoir le réutiliser facilement, ce qui rend vos settings portables d'une machine à une autre. Notez que les settings sont du Python. Vous pouvez donc exploiter sa force pour dynamiser le contenu de ce fichier au besoin.

À ce point de la configuration, vous pouvez analyser le code SQL utilisé lors de la création du modèle avec la commande :

```
$ python manage.py sqlall blog
```

Il ne reste plus qu'à lancer la commande **syncdb** qui permet de générer la base de données :

```
$ python manage.py syncdb
```

Vous devriez normalement voir apparaître :

```
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table blog_post
Creating table django_comments
Creating table django_comment_flags
```

```
You just installed Django's auth system, which means you don't have
any superusers defined.
Would you like to create one now? (yes/no):
```

Il est conseillé à cet instant de répondre *yes* et de créer un compte de super-utilisateur afin de pouvoir accéder à l'interface d'administration plus tard dans le tutoriel. Lorsque c'est fait, la création de la base se poursuit.

Vérifions le bon fonctionnement du nouveau modèle **Post** avec la ligne de commande de Django que l'on appelle avec la commande suivante :

```
$ python manage.py shell
```

On va pouvoir créer un nouveau **Post**, le récupérer et le modifier :

```
>>> from blog.models import Post
>>> post = Post.objects.create(title=u"Premier billet", content=u"Avec du contenu")
>>> Post.objects.all()
[<Post: Premier billet>]
>>> post.title = u"Premier billet 2"
>>> post.save()
>>> Post.objects.all()
[<Post: Premier billet 2>]
```

Il est possible de faire des requêtes beaucoup plus poussées, mais cela dépasse le périmètre de ce tutoriel. N'hésitez pas à vous référer à la documentation officielle pour apprendre à utiliser celles-ci.

Les données étant accessibles, passons à leur affichage qui constitue l'intérêt principal de votre futur visiteur.

1.3 Zope 3

Le mode de stockage par défaut de Zope (la ZODB) est en grande partie responsable de son succès et de sa puissance. Pourtant, c'est un point qui effraie toujours ceux qui ne l'ont pas essayé, car il se situe un peu en marge des pratiques les plus répandues. Contrairement aux bases de données relationnelles comme PostgreSQL ou MySQL qui stockent les données sous forme tabulaire en établissant des relations entre les tables, la ZODB (*Zope Object DataBase*) est une base de données objet : les objets Python sont enregistrés tels quels sur le disque dans une forme sérialisée (le « *pickle* »). Utiliser la ZODB ne nécessite pas de créer des tables et des colonnes à l'avance. Une ZODB vide contient juste un objet racine se comportant comme un **dict**, et implémenté sous forme de BTree pour des questions de performance. On peut placer des objets à l'intérieur de cette racine, comme avec n'importe quel objet **dict** Python, ou bien lui ajouter des attributs arbitraires. Il est ainsi possible de créer une hiérarchie d'objets persistants quelconques et de les utiliser sans se soucier du fait qu'ils sont dans une base de données, ni même avoir à effectuer des requêtes. Pour créer un objet persistant, il suffit de le faire dériver de la classe **Persistent**. Cette souplesse de stockage rappelle la souplesse du typage dynamique de Python : elle offre une liberté énorme, mais nécessite de la discipline pour assurer la cohérence et la pérennité des données.

La ZODB possède des qualités indéniables, qui lui permettent d'être utilisée pour des projets critiques depuis une décennie : elle est transactionnelle (niveau ACID, avec prise en charge des *savepoints*). Elle conserve tout son historique et permet de revenir à n'importe quel moment du passé. Elle gère efficacement les gros objets binaires (*blobs*). Elle autorise plusieurs implémentations de stockage (par défaut un

gros fichier comme une base SQL). Enfin, elle possède un système d'accès réseau concurrent permettant de créer des clusters facilement (ZEO). En outre, elle supporte très bien les gros volumes et il est courant de rencontrer des ZODB stockant plusieurs dizaines de Go.

Au chapitre des performances, la ZODB supporte assez bien la comparaison avec une base de données SQL, même s'il est difficile d'établir des *benchmarks*, tant le mode d'utilisation est différent. Les rares personnes qui ont tenté ce genre d'exercice en essayant de reproduire des conditions équitables ont été surprises par ses performances intrinsèques, qui ne se dégradent que lors de nombreux accès simultanés en écriture. La ZODB est donc adaptée aux applications dont la répartition lecture/écriture est de 80%/20%. Elle est idéale pour gérer des données possédant un haut niveau d'abstraction, plutôt organisées de manière hiérarchique, donc notamment des applications de gestion de contenu (CMS). Par contre, si les données à gérer sont purement tabulaires, une base SQL est souvent plus adaptée.

Vous pouvez utiliser la ZODB de manière indépendante, sans Zope, en gérant vous-même la connexion. Voici comment installer la version 3.8.1 dans un Virtualenv :

```
$ virtualenv zodbtest
$ cd zodbtest
$ easy_install ZODB3==3.8.1
```

Voici ci-dessous un exemple d'utilisation : une base de données est créée avec l'implémentation de stockage **Filestorage**, la plus courante (on pourrait aussi bien utiliser **DirectoryStorage**, qui crée une arborescence de fichiers plutôt qu'un gros fichier **Data.fs** ou bien **RelStorage** qui enregistre les objets dans une base SQL). Ensuite, on définit notre objet à stocker, la classe **Peluche**, qui doit dériver de **Persistent** pour être persistante. Puis, on crée une instance de **Peluche**, on ouvre une connexion, on récupère l'objet racine et on stocke **alain** dans la racine. Pour finir, on ferme la transaction :

```
from ZODB.FileStorage import FileStorage
from ZODB.DB import DB
from persistent import Persistent
import transaction

storage = FileStorage('Data.fs')
db = DB(storage)

class Peluche(Persistent):
    def __init__(self, nom, desc):
        self.nom, self.desc = nom, desc

alain = Peluche(nom=u"Alain", desc=u"La mascotte de l'AFPY")

connexion = db.open()
racine = connexion.root()
racine['alain'] = alain
transaction.commit()
```

Les objets Python « immutables » (ex : **int**, **tuple**) sont directement persistants et pour les objets mutables (**dict**, **list**), il existe une implémentation persistante : **PersistentDict** et **PersistentList**.

Lorsque vous utilisez la ZODB avec Zope, vous n'avez pas besoin de gérer la connexion, ni la transaction : la connexion est déjà ouverte, et une transaction est automatiquement créée pour chaque requête. Si une erreur survient pendant la requête, la transaction est annulée et les données ne sont pas modifiées.

D'une manière indépendante à la ZODB, Zope autorise la définition de modèles de données grâce à des interfaces (**zope.interface**) et des schémas (**zope.schema**). Ces définitions permettent de documenter les objets, de définir des contraintes sur les données, et éventuellement de générer des formulaires de saisie de manière automatique. Voici à quoi ressemble **Alain** lorsqu'on le définit avec des schémas. Si vous avez conservé votre projet du tutoriel, retournez-y et créez le fichier **src/monprojet/peluche.py** suivant :

```
from zope.interface import Interface, implements
from persistent import Persistent
from zope.schema import TextLine

class IPeluche(Interface):
    nom = TextLine(title=u'Nom')
    desc = TextLine(title=u'Description')

class Peluche(Persistent):
    implements(IPeluche)
    nom = desc = None
```

L'interface **IPeluche** sert à définir le modèle de données grâce à un schéma, tandis que la classe **Peluche** est un exemple d'implémentation de cette interface. Vous pouvez alors ajouter les lignes suivantes dans le fichier **configure.zcml** pour faire apparaître **Une Peluche** dans le menu de gauche de la ZMI pour pouvoir ajouter une instance de peluche dans la ZODB :

```
<browser:addMenuItem title="Une Peluche"
                     class=".peluche.Peluche"
                     permission="zope.ManageContent" />
```

Si vous voulez avoir la moindre chance de pouvoir lire les données d'un objet **Peluche**, il faut aussi définir ses protections. Cette configuration (toujours dans **configure.zcml**) indique que l'interface **IPeluche** de la classe **Peluche** est protégée en lecture par la permission **zope.Public** et en écriture par la permission **zope.ManageContent** :

```
<class class=".peluche.Peluche">
  <require permission="zope.Public" interface=".peluche.IPeluche" />
  <require permission="zope.ManageContent" set_schema=".peluche.IPeluche" />
</class>
```

La ZODB est la base de données par défaut de Zope et il n'est pas encore très facile de s'en séparer complètement. Il existe tout de même un paquet tiers appelé **lovely.zetup** qui fournit un exemple de Buildout contenant une application Zope configurée sans ZODB. Sinon, il est tout à fait possible et même assez courant d'utiliser une base de données relationnelle, soit directement en Python, soit en utilisant un ORM comme SQLAlchemy, tout en profitant des transactions et des schémas de Zope (voir les paquets **zope.sqlachemy**, **z3c.dobbin**, **ore.alchemist**).

2 Vues, templating et présentation

Dans un framework web, on n'affiche jamais les données de manière directe. Celles-ci sont transmises à des composants qui se chargent de l'affichage. Il s'agit de la partie « V » du MVC, c'est-à-dire la vue. On assiste à un glissement sémantique de la notion de vue selon le framework : les *views* sont des fonctions dans Django, des classes dans

Zope et des *templates* dans Pylons. Tandis que l'équivalent des views de Django et Zope, c'est-à-dire le code Python qui fait le lien entre le modèle et la vue, porte le nom de *controller* dans Pylons, qui est le plus proche du modèle MVC classique.

Quels que soient les points de vue, on retrouve toujours en bout de chaîne un système de templating. Cette technique permet de créer des squelettes (ou gabarits) de page HTML contenant des instructions simples pour inclure des données en provenance de vos classes Python (les contrôleurs de Pylons ou les vues de Django et Zope), mais aussi des boucles, des conditions, etc. On voit s'affronter deux tendances : le templating XML (**zpt**, **genshi**, **chameleon**, **z3c.pt**) et le templating à balises (**mako**, **jinja**, **django**, **cheetah**). Le premier est généralement plus lent, mais les gabarits de page sont des documents XML ou XHTML valides qu'on peut afficher tels quels dans un navigateur et même transmettre à un graphiste qui pourra faire son travail sans déranger les développeurs. Le deuxième fait appel à des balises spécifiques utilisant des accolades ou des chevrons qui empêchent de vérifier a priori la validité finale du document, mais il s'avère souvent plus rapide, et peut être utilisé pour générer autre chose que des pages web, par exemple des fichiers texte ou des documents PDF. Certains systèmes de templating autorisent ou interdisent l'inclusion de code Python directement dans le gabarit. D'autres peuvent être compilés en *bytecode* Python pour décupler les performances.

2.1 Pylons

Pylons laisse le choix du langage de templating au concepteur de l'application.

Vous pouvez utiliser **Mako**, **Genshi** ou **Jinja** qui sont tous les trois proposés lors de la création d'un projet. Intégrer un autre moteur de templating dans Pylons se fait en quelques lignes de code sans modifier vos contrôleurs. Il suffit de modifier l'alias **render** vers une autre fonction. La ligne suivante est utilisée par défaut (dans **lib/base.py**) et illustre le principe. C'est cette ligne qu'il faudra modifier pour utiliser un autre moteur de templating :

```
from pylons.templating import render_mako as render
```

Cette méthode **render** s'utilise ensuite en fournissant juste le fichier de gabarit **index.mako**, situé dans le dossier **monprojet/templates** :

```
def index(self):
    render('/index.mako')
```

Mako est un langage de templating souple qui supporte les imports, les fonctions et les macros. Notre gabarit de page pourrait par exemple ressembler à ceci :

```
# -*- coding: utf-8 -*-
<%!
from datetime import datetime
%>\
<% now = datetime.now() %>
<html>
<body>
<h1>Youhou !</h1>
<p>Que fais-tu debout à cette heure ??!. Il n'est que ${now.strftime('%H')}h</p>
</body>
</html>
```

Dans la réalité, il est souvent nécessaire de fournir un gabarit maître fournissant la structure de base, pour éviter d'avoir à la répéter dans tous les autres. Avec **Mako**, le gabarit maître peut ressembler à ceci (fichier **base.mako**) :

```
## -*- coding: utf-8 -*-
<!DOCTYPE H™L PUBLIC "-//W3C//DTD H™L 4.01 Transitional//EN" "http://
www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
<title>Youhou !!</title>
</head>
<body>
  <h1>Youhou !</h1>
  ${self.body()}
</body>
</html>
```

Et notre page de contenu peut maintenant faire appel au gabarit maître de cette façon :

```
## -*- coding: utf-8 -*-
<%inherit file="/base.mako" />\
<%!
from datetime import datetime
%>\
<% now = datetime.now() %>
<p>Que fais-tu debout à cette heure ??!. Il n'est que ${now.strftime('%H')}h</p>
```

2.2 Django

Les templates constituent la raison d'être de votre application, comme nous l'avons vu dans le guide de démarrage. Django utilise un langage permettant de faciliter la gestion des inclusions de variables dynamiques.

Dans le cadre de notre exemple de blog, nous allons avoir besoin de deux vues, celle affichant la liste des billets et celle affichant un billet unique. Commençons par la liste. Comme nous allons utiliser les vues génériques, le nom du template sera par convention **blog/post_list.html**, à créer dans le dossier **templates** qui a été défini dans le tutoriel de démarrage. Nous allons afficher dans ce template la liste des titres des billets présents :

```
{% extends "base.html" %}

{% block content %}
<ul>
{% for object in object_list %}
    <li>
        <a href="{% url blog_post object.id %}" title="{{ object.title }}">
            {{ object.title }}
        </a>
    </li>
{% endfor %}
</ul>
{% endblock %}
```

Plusieurs choses en très peu de lignes :

- Par convention, les templates héritent généralement d'un template **base.html** qui définit des blocs ré-exploitables comme ici avec le bloc content. Ce template pourrait par exemple contenir :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XH™L 1.Ø Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
    {% block sidebar %}{% endblock %}
</body>
</html>
```

- Pour boucler sur les objets de type **Post**, on utilise la boucle **for** du langage de templates de Django, son usage étant très proche de la syntaxe Python : les instructions sont contenues dans **{%%}** et les variables entre **{{}}**.
- Il est possible d'accéder directement aux attributs des modèles et des dictionnaires dans les templates en utilisant le « **.** » (point) comme ici avec **post.title**.
- Le tag **url** permet de récupérer automatiquement l'adresse d'un objet si le nom a été défini dans les URL comme nous le verrons dans la partie dédiée.

Ce template devrait nous afficher la liste des titres de billets au même titre que le suivant (à sauver sous **blog/post_detail.html**) devrait nous afficher le billet en question lorsqu'on clique sur le lien :

```
{% extends "base.html" %}

{% block content %}
<h1>{{ object.title }}</h1>
<div>{{ object.content }}</div>
{% endblock %}
```

Les templates sont en place. Il est temps de passer à la logique qui contrôle la mise à disposition des variables utilisées dans les templates en fonction des URL appelées : les vues.

2.3 Zope 3

Le principe de Zope est de publier des objets. Les objets, enregistrés dans la ZODB, ne sont pas visibles directement. Il est donc nécessaire de créer des vues, qui sont des classes Python dont le rôle est de préparer les données afin d'être affichées dans une page web. Ces classes Python, qu'on appelle vues en langage Zope, correspondent en fait aux contrôleurs dans le modèle MVC. Les vues ont accès à deux objets : l'objet contexte (celui qu'on veut afficher), et l'objet requête (provenant du navigateur). En sortie, la vue fournit une page HTML, en utilisant éventuellement un gabarit de page (un template). Ceci est le cas le plus classique (un site web publié par HTTP). Il est possible de publier le même objet en utilisant un autre protocole, comme FTP ou XML-RPC. L'objet ne change pas, mais la vue permet de « voir » cet objet différemment. Zope choisit la bonne vue en fonction de plusieurs paramètres : son nom (par exemple **"index.html"**), l'objet qu'on veut voir, et l'objet requête. Ces informations constituent la « configuration » de la vue, qui se matérialise par une inscription dans le registre de composants en utilisant le langage ZCML. Voici tout d'abord un exemple de vue pour notre classe **Peluche**. Vous pouvez ajouter le code suivant à la suite du fichier **peluche.py** (en général, on le met dans un fichier **browser.py**) :

```
from zope.publisher.browser import BrowserView

class PelucheView(BrowserView):
    titre = u"Une vue de l'objet peluche"
```

Puis, créez le gabarit (le template) HTML dans le fichier **peluche.pt** :

```
<html><body>
  <h1 tal:content="view/titre">un titre</h1>
  Nom de la peluche : <span tal:content="context/nom">nom_peluche</span><br/>
  Description : <span tal:content="context/desc">nom_peluche</span>
</body></html>
```

Le templating de Zope s'appelle ZPT. Il est de type XML. Il remplace l'ancien système (DTML) qui était un langage à balises. Les gabarits sont donc au format XML et les commandes pour inclure du contenu prennent la forme d'attributs XML dans l'espace de nom TAL (*Template Attribute Language*), contenant des expressions avec la syntaxe TALES (*TAL Expression Syntax*). La définition paraît complexe, mais l'utilisation est simple : dans le titre **h1** ci-dessus, la directive TAL est **content**, l'expression TALES est **view/titre**. Elle commande de remplacer le contenu de la balise **h1** par l'attribut nom du contexte (l'objet publié). Le faux titre **un titre** sert d'exemple lors de la visualisation du gabarit tel quel dans un navigateur. D'autres directives permettent de faire des boucles, des conditions ou de modifier un attribut XML.

Enfin, configurez la vue en définissant son nom, sa permission, son gabarit et les types d'objets auxquels elle s'applique (dans **configure.zcml**) :

```
<browser:page name="view.html"
              class=".peluche.PelucheView"
              for=".peluche.IPeluche"
              template="peluche.pt"
              permission="zope.Public" />
```

Après redémarrage, si vous avez ajouté un objet peluche depuis la ZMI (par exemple **'alain'**), vous pouvez observer **alain** au travers de la vue **view.html** à l'adresse http://localhost:8000/alain/view.html. Vous pouvez constater qu'**alain** n'a pas encore de nom, ni de description. Pour modifier **Alain**, nous avons besoin d'un formulaire web.

On a vu un peu plus haut que les modèles de données sont décrits par des interfaces ou des schémas, embarquant des informations et des contraintes. Ces informations sont en réalité suffisantes pour générer automatiquement un formulaire web ! Cette génération est prise en charge au choix par les paquets **zope.formlib** ou **z3c.form**. Ces bibliothèques (surtout la deuxième) offrent un très haut niveau de modularité et autorisent la personnalisation de toutes les étapes de validation, de conversion et d'affichage des données à l'aide de *widgets*. Voici un exemple de formulaire autogénéré pour notre peluche. Ajoutez la vue suivante dans **peluche.py** :

```
from zope.formlib.form import EditForm, Fields

class PelucheEdit(EditForm):
    form_fields = Fields(IPeluche)
```

Puis, ajoutez la configuration suivante dans **configure.zcml** :

```
<browser:page name="edit.html"
              class=".peluche.PelucheEdit"
              for=".peluche.IPeluche"
              permission="zope.Public" />
```

Après redémarrage, vous disposez d'un formulaire à l'adresse http://localhost:8080/alain/edit.html pour modifier votre peluche. Vous pouvez ensuite retourner à la vue **view.html** pour vérifier que vos données ont été enregistrées. Le formulaire conserve l'apparence de la ZMI, car, dans cet exemple, aucun gabarit de contenu ni de gabarit de mise en page ne lui a été attribué.

Pour la mise en page et le *skinning*, de nombreuses solutions sont disponibles. Par défaut, un système de macros (METAL) permet de créer des héritages de gabarits. On peut donc préparer un gabarit à trous (des *slots*) qui fournit le style et la mise en page. Chaque gabarit de contenu pourra boucher les trous en utilisant des directives XML **fill-slot**. L'intérêt est que le gabarit maître et les gabarits de contenu sont tous des fichiers XHTML complets.

Les *content providers* et les *viewlets* sont une façon de gérer un morceau de page web de manière indépendante,

grâce à une classe ayant accès au contexte, à la requête et à la vue. Ils sont utiles pour créer des colonnes, des boîtes d'information ou des menus variables en fonction du contexte ou du visiteur.

Les *pagelets* sont des vues dont le principe est de rendre complètement indépendants la vue, son gabarit, et le gabarit de mise en page. Chacun des trois ne contient aucune information sur les deux autres, et le câblage est effectué par de la configuration (en ZCML). Il est ainsi possible de changer arbitrairement de gabarit ou une mise en page à n'importe quel endroit. Les pagelets permettent de créer des gabarits réutilisables et de les distribuer sans faire de supposition sur leur utilisation. C'est une tentative réussie d'étendre la notion de réutilisabilité à la couche de présentation d'une application web.

Pour finir, Zope 3 offre un support complet de l'unicode. La traduction d'une application se fait grâce à la bibliothèque **gettext** et un outil permet d'extraire les chaînes à traduire dans le code Python, les gabarits de page et les fichiers de configuration ZCML. Par défaut, le langage utilisé change en fonction de votre navigateur, et on peut forcer la langue par différents moyens.

3 Contrôle de la logique applicative

Le « C » du MVC, c'est-à-dire le Contrôleur est peut-être le plus difficile à définir. On peut le situer généralement dans des classes Python que le développeur doit écrire, parfois dans la gestion des URL ou même dans l'ensemble des mécanismes offerts par le *framework*. Quoi qu'il en soit, il héberge une bonne partie de la logique de votre application et régule les interactions entre les Modèles, les Vues, et l'utilisateur.

3.1 Pylons

Pylons est un pur framework MVC : pour faire le lien entre les modèles et les vues (les gabarits de page), il utilise des contrôleurs qui sont des classes Python. Les méthodes de ces classes contrôleurs correspondent à des actions de l'utilisateur. Elles peuvent utiliser les modèles (la couche « M ») pour réaliser l'action correspondante, puis renvoyer un résultat à l'utilisateur.

Un contrôleur permet de regrouper des actions dont le but est proche. Nous allons créer un contrôleur pouvant lire et écrire dans un simple fichier texte. Pour démarrer rapidement, l'outil Paste offre le moyen de créer un squelette de contrôleur (et de son test) de la manière suivante :

```
$ paster controller fichiertexte
```

Nous pouvons maintenant éditer le fichier **fichiertexte.py** et modifier le contrôleur :

```
class FichiertexteController(BaseController):

    fichier = '/tmp/fichier.txt'

    def lire(self):
        return open(self.fichier).read()

    def ecrire(self):
        if request.POST and 'contenu' in request.POST:
            contenu = request.POST.get('contenu')
            open(self.fichier, 'w').write(contenu)
            redirect_to('lire')
        return render('/formulaire.mako')
```

Il faut également créer un gabarit **formulaire.mako** dans le dossier **templates** :

```
<html><body>
  <form action="ecrire" method="post">
    <textarea name="contenu"></textarea>
    <button type="submit">sauver</button>
  </form>
</body></html>
```

Le contrôleur comporte deux actions : une pour lire le fichier texte et le renvoyer à l'utilisateur, une autre pour écrire un fichier texte depuis des données envoyées via un formulaire. L'action **ecrire** est disponible à l'adresse http://localhost:5000/fichiertexte/ecrire sans qu'il y ait besoin d'effectuer la moindre configuration. Si le formulaire est validé, l'objet requête contient des données **POST** utilisées pour créer le fichier texte. Ensuite, le contrôleur redirige vers l'adresse de l'action lire.

Cet accès direct depuis l'URL jusqu'aux contrôleurs et aux actions est une fonctionnalité offerte par le paquet **Routes**. **Routes** est une réécriture en Python du composant éponyme de Ruby On Rails. Il permet de configurer avec précision le routage des URL vers le code Python de vos contrôleurs. Une route du type **/{controller}/{action}** est déjà présente par défaut. C'est justement celle qui permet de faire aboutir l'URL **/fichiertexte/ecrire** à la bonne action du bon contrôleur. Vous pouvez modifier les routes et en ajouter à volonté dans le fichier **config/routing.py**. Le paquet **Routes** autorise toutes sortes de routage de manière très souple et permet de créer des URL compréhensibles et efficaces pour le référencement. Les éléments d'une route peuvent être aussi bien statiques, que dynamiques, utiliser des expressions rationnelles, des fonctions filtres, des conditions, des valeurs par défaut. **Routes** permet non seulement l'analyse des URL, mais également leur génération : dans notre action **ecrire**, pour bien faire, il faudrait générer l'URL de redirection en utilisant : **routes.url_for(action='lire')**.

3.2 Django

Il y a deux concepts à comprendre à ce niveau. D'une part, le routage qui consiste à lancer la bonne fonction en fonction de l'URL appelée. Et, d'autre part, la fonction en elle-même qui a pour nom **view** en Django, ce qui peut prêter à confusion, car elle correspond à la partie C du modèle MVC (et non V).

Dans le cadre de ce tutoriel, nous allons utiliser les vues génériques qui permettent de développer plus rapidement. Les vues traditionnelles (comme nous l'avons vu avec l'application **hello_world**) seront donc laissées à titre d'exercice pour enrichir l'existant (il est possible d'étendre les vues génériques comme vous pouvez le lire sur le blog de James Bennett : http://www.b-list.org/weblog/2006/nov/16/django-tips-get-most-out-generic-views/).

Si l'on reprend le fichier des URL du projet de test, on va pouvoir ajouter les nouvelles URL qui permettent de consulter la liste des billets publiés et d'un billet unique :

```
from django.views.generic.list_detail import object_list, object_detail
from hello_world.views import hello
from blog.models import Post

post_dict = {'queryset': Post.objects.all()}

urlpatterns = patterns('',
    (r'^hello_world/$', hello),
    url(r'^blog/(?P<object_id>\d+)/$',  object_detail, post_dict, name="blog_post"),
    url(r'^blog/$',                     object_list, post_dict, name="blog_list"),
)
```

On commence par définir un dictionnaire **post_dict** qui contient une *queryset* au sens Django du terme, c'est-à-dire un itérateur sur des instances de **Post** qui sont récupérées dans la base de données comme nous l'avons vu dans la partie relative aux modèles. Ce dictionnaire est utilisé par les vues génériques pour déterminer à quels types d'objets elles sont appliquées.

On déclare ensuite les deux lignes permettant d'accéder à un billet unique qui aura une URL de type **/blog/1/** ou à la liste des billets sur **/blog/**. L'utilisation de la fonction **url** permet d'utiliser un argument **name** pour attribuer un nom à un pattern donné que nous réutiliserons ensuite.

Vous pouvez facilement tester en relançant le serveur de test :

```
$ python manage.py runserver
```

Et en vous rendant à l'adresse http://127.0.0.1:8000/blog/, vous devriez découvrir la liste des billets. Si vous avez suivi les tests à effectuer dans un shell dans la partie modèles, il devrait y avoir un élément existant intitulé « Premier billet 2 » avec un lien permettant d'accéder au billet.

Tout fonctionne bien ? Parfait, passons maintenant à l'ajout de commentaires.

Il va falloir ajouter une ligne aux URL du projet :

```
(r'^comments/', include('django.contrib.comments.urls')),
```

Les applications tierces possèdent leur propre fichier d'URL que nous incluons ici sous la racine **/comments/**. Une bonne pratique Django est d'avoir un fichier d'URL par application (dans un souci de concision, ce n'est pas fait dans ce tutoriel).

On va maintenant ajouter la liste des commentaires et le formulaire qui permettra de poster un commentaire au template existant **post_detail.html** :

```
{% extends "base.html" %}
{% load comments %}

{% block content %}
<h1>{{ object.title }}</h1>
<div>{{ object.content }}</div>

<h2>Commentaires</h2>
{% get_comment_list for object as comment_list %}
{% for comment in comment_list %}
    <p>Commentaire de {{ comment.user_name }}
    <div>{{ comment.comment }}</div>
{% endfor %}

<h2>Poster un commentaire</h2>
{% render_comment_form for object %}

{% endblock %}
```

On commence par charger avec le mot-clé **load** les outils permettant d'utiliser les commentaires (grâce auxquels on va pouvoir utiliser par la suite les commandes **get_comment_list** et **render_comment_form**).

On récupère ensuite la liste des commentaires existant **comment_list** pour le billet sur lequel on se trouve (**object**) que l'on affiche en bouclant dessus. On utilise aussi un autre outil **render_comment_form** qui permet d'afficher le formulaire standard. C'est suffisant pour tester vos nouveaux commentaires. Rendez vous sur la page du billet précédemment créé. Vous devriez voir le formulaire de commentaire apparaître. Il suffit alors de le remplir et de soumettre votre commentaire.

Par défaut, vous allez être redirigé sur une page vous remerciant pour votre commentaire. Il serait intéressant d'ajouter un lien permettant de revenir au billet qui vient d'être commenté. Il suffit pour cela de définir votre propre template **comments/posted.html** dans le dossier **templates** et d'y inclure les lignes suivantes :

```
{% extends "comments/base.html" %}

{% block content %}
<h1>Merci pour votre commentaire.</h1>
<p>
    Revenir au billet
    <a href="{% url blog_post comment.content_object.id %}" title="">
        {{ comment.content_object.title }}
   </a>
</p>
{% endblock %}
```

Django commence par chercher un template donné dans votre dossier de templates avant d'essayer de le trouver dans les dossiers des applications. Il est ainsi très facile, comme nous venons de le voir, de personnaliser le rendu d'applications existantes.

L'objet **comment** est présent dans le contexte du template et il est possible d'accéder au billet commenté avec le champ **content_object** qui est ici utilisé pour afficher le titre et faire un lien vers le billet en question. Il est rendu possible d'utiliser le *templatetag* **url** grâce à l'URL nommée (**blog_post**) définie précédemment.

Vous disposez maintenant d'une interface utilisateur fonctionnelle, mais il serait intéressant de pouvoir ajouter de nouveaux billets avec une interface plus intuitive qu'un terminal ! C'est l'objet de la partie gestion que nous allons prochainement aborder.

3.3 Zope 3

S'il fallait faire correspondre le modèle MVC à Zope, le rôle du Contrôleur pourrait être attribué en partie aux *browser views* (voir chapitre précédent) qui sont des classes Python similaires aux contrôleurs de Pylons ou aux fonctions-vues de Django. Néanmoins, le rôle le plus important doit être attribué à la Component Architecture qui est la plus grosse particularité de Zope et qui fait l'objet d'un article séparé dans le hors-série n°40 dédié à Python. Le principe de cette architecture est de se baser exclusivement sur des fonctionnalités décrites par des interfaces. En outre, ces interfaces ne sont, la plupart du temps, pas offertes par les objets de contenu eux-mêmes, mais par des adaptateurs.

Prenons l'exemple de la taille d'un objet. Celle-ci peut être définie de multiples façons. Pour un conteneur, il peut s'agir du nombre d'objets contenus, et, pour une image, de son poids en octets ou bien de ses dimensions. L'interface permettant d'accéder à la taille d'un objet est

ISized. **ISized** offre deux méthodes : **sizeForDisplay()** et **sizeForSorting()**. Les objets eux-mêmes n'implémentent pas cette interface. Par contre, il est possible d'écrire des adaptateurs offrant cette interface, et pouvant s'appliquer à des types d'objets précis. On peut écrire un adaptateur offrant **ISized** pour une image, un autre adaptateur offrant **ISized** pour un conteneur, et un adaptateur par défaut essayant diverses méthodes pour calculer une taille. Ci-dessous, nous allons créer un exemple d'adaptateur de taille pour notre fameuse peluche **Alain**, afin de pouvoir calculer la taille d'une peluche de manière générique, sans jamais toucher à notre implémentation de **Peluche**. Vous allez voir que l'opération fera apparaître la taille de la peluche dans l'interface de la ZMI. Pour l'instant, vous devez voir apparaître « non disponible » dans la colonne « Taille ». Ajoutez les lignes suivantes dans **peluche.py** :

```
from zope.component import adapts
from zope.size.interfaces import ISized

class PelucheSize(object):
    implements(ISized)
    adapts(IPeluche)

    def __init__(self, context):
        self.context = context

    def sizeForSorting(self):
        return len(self.context.desc or '')

    def sizeForDisplay(self):
        return unicode(len(self.context.desc or '')) + u' caract.'
```

Cet adaptateur calcule (bêtement) la taille d'une peluche en comptant le nombre de caractères de l'attribut **desc**. Il implémente l'interface **ISized** (au travers des deux méthodes) et s'adapte à tous les objets fournissant **IPeluche**. L'objet adapté s'appelle le contexte, il est récupéré grâce au constructeur **__init__** et peut être utilisé à toutes fins utiles. Pour inscrire cet adaptateur dans le registre de composants et le rendre disponible au framework, il suffit de rajouter cette ligne dans **configure.zcml** :

```
<adapter factory=".peluche.PelucheSize" />
```

Après redémarrage, vous devriez voir apparaître la taille de votre peluche dans la ZMI.

À toutes les étapes d'un traitement, le framework effectue des *lookup* dans le registre de composants, afin d'obtenir le composant souhaité (la plupart du temps un adaptateur). Ces lookup sont extrêmement rapides, car le registre est créé une fois au démarrage du serveur, puis conservé en mémoire.

La logique applicative dans Zope s'écrit donc en général de manière complètement générique, en se basant exclusivement sur des interfaces correspondant à des fonctionnalités. Il est ainsi possible de découper proprement une application en fonctionnalités, en permettant à celles-ci de s'appliquer à n'importe quel objet. Chaque fonctionnalité peut ensuite être réutilisée dans une autre application.

La gestion des URL n'échappe pas à la règle. L'évaluation d'une URL par le framework se fait de gauche à droite, en partant de la racine, et chaque objet décide de ce que représente pour lui la suite de l'URL, en utilisant l'interface **IPublishTraverse**. Prenons l'URL http://foo/bar/baz.html : le framework interroge le registre pour trouver un adaptateur de l'objet racine fournissant **IPublishTraverse**. L'adaptateur retourne l'objet correspondant à la chaîne de caractère **'foo'** vis à vis de la racine. Ensuite, le framework interroge le registre pour trouver un adaptateur de l'objet **foo** fournissant **IPublishTraverse**. Ce nouvel adaptateur renvoie l'objet correspondant à **'bar'** vis à vis de l'objet **foo**. Enfin, de la même façon, le registre récupère l'objet correspondant à **'baz.html'** vis à vis de l'objet **'bar'**. Il se trouve que cet objet est une vue, publiable, donc capable de renvoyer une page HTML, ce qu'elle fait. Pour une application de gestion de contenu, les objets sont souvent des conteneurs, et le *traversing* d'un objet revient à récupérer l'objet contenu. Mais, on peut gérer aussi bien des URL de type **blog/2008/12/02**, où **2008**, **12** et **02** ne sont pas des objets, mais juste des informations permettant de rechercher le bon article de blog dans la ZODB ou dans n'importe quelle autre source de données, base SQL, fichier texte ou même une source différente pour chaque type d'objet. C'est à l'adaptateur de décider où il récupère l'objet.

Zope fait donc l'objet de ce qu'on pourrait appeler une « inversion de contrôle » : dans un framework MVC classique, l'analyse de l'URL aboutit d'abord à un Contrôleur, puis le Contrôleur s'adresse éventuellement au Modèle, et renvoie vers la Vue. Dans Zope, l'analyse de l'URL, au lieu d'aboutir à un Contrôleur, aboutit d'abord au Modèle, par défaut, un objet dans la ZODB, puis le Contrôleur-Vue est choisi en fonction du Modèle, grâce à un lookup dans le registre de composants. Ceci facilite les comportements dépendants du contenu. Pour une même URL, on pourra obtenir des comportements ou des vues radicalement différents selon le type de l'objet, sans écrire une ligne de code.

4 Sécurité et comptes utilisateurs

La grande majorité des applications ou sites webs possèdent des comptes utilisateurs. Les besoins sont très variés, allant d'un petit site web ne nécessitant qu'un compte administrateur à un gros intranet d'entreprise ayant besoin d'une gestion complète des groupes, rôles ou profils et permissions, avec possibilité de délégation, authentification par LDAP, OpenId ou depuis des sources variées. Ces fonctions font partie des besoins de base au même titre que le templating et tous les frameworks proposent leur propre solution, soit intégrée, soit grâce à des outils externes. Sur ce dernier point, on note une tendance à déporter l'authentification dans des middlewares WSGI afin d'unifier l'authentification de plusieurs applications, par exemple un *wiki*, un CMS et un outil de suivi de bogues.

4.1 Pylons

Pylons n'offre par défaut aucune gestion des droits d'accès, ni de l'authentification. Fidèle à son principe, il laisse au développeur le soin d'implémenter lui-même la sécurité ou d'utiliser un outil tiers, plus particulièrement un middleware WSGI dédié à cette tâche. Pylons lui-même se contente simplement d'offrir des points d'ancrage sur les contrôleurs à l'aide de deux règles simples : les méthodes de contrôleurs commençant par un tiret bas (« _ ») sont privées et ne peuvent pas être accessibles par l'URL ; et la méthode **__before__** de chaque contrôleur est appelée avant

toute autre méthode. L'idée de base est donc d'effectuer les tâches à sécuriser dans les méthodes privées, d'y accéder depuis des méthodes publiques en contrôlant l'accès dans la méthode `__before__`.

Pour les cas avancées, les deux composants les plus utilisés sont, au choix, `Authkit` ou `repoze.who`. Le premier des deux se compose de trois parties : un middleware d'authentification, des objets représentant les permissions, et des adaptateurs d'autorisation (décorateurs à poser sur les actions, middleware d'autorisation). Le deuxième, `repoze.who`, provient de la communauté Zope et de son ouverture à WSGI. Il rencontre actuellement un franc succès et est adopté aussi par TurboGears. Il s'inspire du composant Pluggable Authentication Service de Zope 2 et apporte un lot de *plugins* et une modularité à tous les niveaux. `Authkit` et `repoze.who` possèdent tous les deux leur gestion des rôles, des groupes et des permissions.

Pour ce qui est des protection de sécurité, elles sont fournies individuellement par les différents outils tiers utilisés : par exemple, SQLAlchemy vous protège entre autres contre les injections SQL en échappant tous les caractères sensibles dans vos données. Pour subir une attaque de ce type, il ne suffira pas de fauter par omission, mais vous devrez implémenter vous-même, et volontairement, la faille de sécurité.

4.2 Django

Django dispose d'un module d'authentification qui permet nativement de gérer des comptes utilisateurs et leurs permissions. Il est possible d'étendre facilement ce système de permissions si vous souhaitez ajouter des permissions métier qui vous sont propres. Le framework dispose aussi d'une gestion native des groupes d'utilisateurs auxquels vous pouvez attribuer des permissions.

Un système de profils existe pour étendre le modèle de vos utilisateurs, il est aussi possible depuis Django 1.0 de sous-classer le modèle par défaut afin d'ajouter des champs et/ou fonctions.

La restriction d'accès à des vues se fait grâce à des décorateurs (`login_required` ou `permission_required` par exemple), mais vous pouvez facilement opter pour une gestion plus fine des droits au sein même de vos vues si vous souhaitez restreindre les objets présentés en fonction d'un utilisateur donné. Il est aussi possible de restreindre l'accès à des données via les managers, ce qui constitue une sécurité intéressante si vous êtes plusieurs à participer à un projet.

Au chapitre de la sécurité, il est à noter que Django vous protège des injections SQL et des injections par les *headers* des emails, des failles de *Cross-Site Scripting* (XSS), *Cross-Site Request Forgery* et *Session Forging/Hijacking* de manière native, si vous n'essayez pas de contourner intentionnellement ces protections. Par défaut, les variables affichées dans les templates sont échappées afin de réduire les risques encourus par les personnes ne maîtrisant pas le sujet.

Le chapitre du Django Book sur la sécurité est une référence dans le domaine : http://www.djangobook.com/en/1.0/chapter19/ (traduit sur http://djangobook.zindep.com/1.0/chapitre19/)

4.3 Zope 3

Le principe adopté dans Zope 3 est de considérer les droits d'accès comme de la configuration et de la séparer complètement du code Python pour la déporter dans des déclarations externes. Les composants métier n'embarquent donc aucun code spécifique à Zope, et n'ont aucune notion des permissions. Il n'est quasiment pas nécessaire de s'en préoccuper lors de leur développement et de leur test. Lors de leur utilisation, les objets sont enrobés automatiquement par des proxys de sécurité qui les protègent de manière transparente pour l'application. Cependant, le comportement par défaut est de tout bloquer. La configuration des permissions est donc un passage obligé pour voir apparaître la moindre page de contenu.

Les composants Python étant considérés comme des boîtes noires pilotables au travers d'une ou plusieurs interfaces, les déclarations de sécurité portent exclusivement sur les interfaces ! Prenons l'exemple d'un objet conteneur : la classe `Folder` implémente l'interface `IContainer` et cette interface est subdivisée en `IReadContainer`, offrant les méthodes de lecture et `IWriteContainer`, offrant les méthodes d'écriture. Il est possible de séparer les droits et de poser des permissions différentes sur ces deux interfaces, pour permettre à certains de lire et à d'autres d'écrire. Les déclarations se font à l'aide de permissions, et les permissions sont regroupées en rôles. Les utilisateurs (ou groupes d'utilisateurs) peuvent ensuite posséder un ou plusieurs rôles. Chose notable dans Zope 3, les rôles et les permissions sont des *utilities* (voir l'article consacré à la Zope Component Architecture). Ils peuvent donc être globaux à l'application ou locaux, c'est-à-dire stockés en base de données et dépendant de l'emplacement dans la hiérarchie des objets. De cette façon, un utilisateur pourra avoir un rôle dans un dossier et un autre rôle dans un autre dossier. Dans le contexte d'une application de gestion de contenu, la délégation devient facile à mettre en place.

Les déclarations de sécurité peuvent s'appliquer à tous les niveaux, depuis les objets persistants jusqu'aux vues, en passant par les adaptateurs intermédiaires. On peut donc protéger un objet différemment de sa vue, et laisser celle-ci afficher un texte différent selon qu'elle peut ou non accéder à l'objet, tout en la rendant inaccessible à d'autres utilisateurs.

L'authentification est isolée dans un composant dont l'unique but est de fournir l'interface `IAuthentication`. Le composant fourni par défaut, `PluggableAuthentication`, sépare clairement la partie « demande d'identité » (en anglais *challenge*, mettant en jeu des *credentials*) de la partie « vérification d'identité ». Ces deux parties sont déléguées à des modules configurables. On peut ainsi effectuer la demande d'identité à la session, puis, si la session ne contient rien, demander un identifiant et un mot de passe. La vérification d'identité peut ensuite être effectuée consécutivement auprès d'un dossier local de la ZODB, puis par exemple dans un annuaire LDAP et enfin dans une base SQL.

5 Gestion et administration

La notion d'administration peut prendre plusieurs formes : soit des outils en ligne de commande, soit une interface web, plus rarement un client lourd. Cette interface peut avoir des buts variés (gestion du code, du serveur, de la base

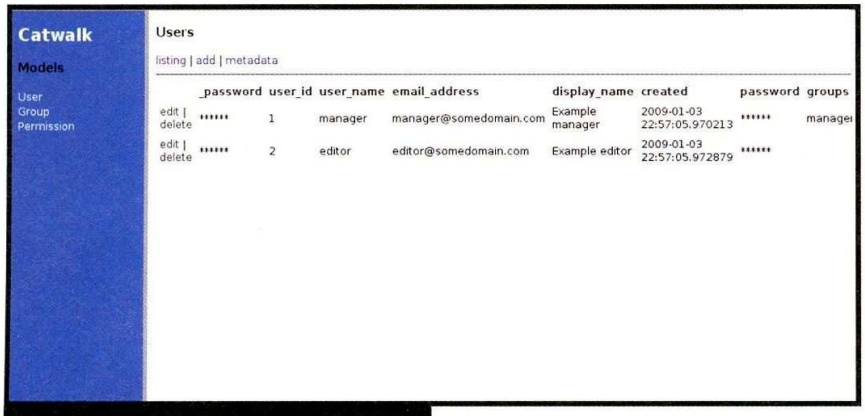
de données, des utilisateurs) et elle peut potentiellement s'adresser à différents publics : des développeurs, des intégrateurs, des administrateurs ou des utilisateurs.

Parler d'interface d'administration pour un framework mène inévitablement à établir une analogie avec les CMS : le constat est que la frontière entre les frameworks et les CMS ne cesse de se réduire. Un CMS est une application de gestion de contenu destinée à des utilisateurs finaux, mais sur lequel un développeur (ou au moins un intégrateur) va passer un temps non négligeable. Un framework est au contraire une application destinée avant tout à des développeurs, mais auxquels les utilisateurs finaux seront tôt ou tard confrontés au travers de l'application développée. L'interface d'administration aurait pu être un élément différenciateur, mais ce n'est même plus le cas : certains frameworks fournissent une interface d'administration qui peut, avec plus ou moins d'effort, se transformer en console de gestion d'un site web. À l'inverse, certains CMS comme Plone sont de réelles plateformes de développement et peuvent quasiment porter le titre de framework web...

5.1 Pylons

La seule interface de gestion fournie par défaut avec Pylons est l'outil Paster en ligne de commande. Celui-ci est un peu un outil à tout faire, puisqu'il permet de créer un squelette de projet ou un squelette de contrôleur, de démarrer ou arrêter le serveur, d'initialiser l'application et sa base de données, et il peut être étendu à volonté en créant des commandes personnalisées pour n'importe quel usage, par exemple lancer des tests de qualité sur le code.

Lorsqu'on parle d'interface d'administration, il s'agit normalement de l'administration des données stockées dans la base. Comme Pylons ne propose pas de méthode unique de stockage, il ne peut pas non plus proposer d'interface graphique pour les administrer et ce n'est pas son rôle. Néanmoins, il existe des solutions. La première est évidemment de jeter un œil à TurboGears ! TurboGears est un framework dont le but est de proposer un ensemble cohérent et intégré, construit par assemblage des meilleurs outils disponibles. Pylons fait justement partie de ces outils et constitue même la base de TurboGears 2.0, qui devrait être publié incessamment. Plusieurs autres bibliothèques conseillées pour Pylons, comme **repoze.who** ou SQLAlchemy, font aussi partie de TurboGears 2. L'intérêt de TurboGears est, entre autres, de fournir une interface d'administration, construite à l'aide de CatWalk. Catwalk est une sorte de phpMyAdmin vous permettant toutes les opérations « CRUD » (**Create**, **Read**, **Update**, **Delete**) à partir de vos modèles de données SQLAlchemy. Voici à quoi ressemble CatWalk juste après installation de TurboGears 2 :



Administration SQLAlchemy avec CatWalk

Si vous voulez utiliser Pylons seul avec SQLAlchemy, une autre solution est d'essayer FormAlchemy. FormAlchemy est une bibliothèque de génération de formulaires à partir des modèles SQLAlchemy. Il suffit de lui fournir la classe du modèle et il se charge du reste. La dernière version (1.1) de Formalchemy contient une extension pour Pylons dont le but est le même que CatWalk : effectuer toutes les opérations de base sur les modèles.

5.2 Django

Un des principaux avantages à utiliser Django est son interface d'administration auto-générée. En quelques lignes, vous allez pouvoir générer une interface web à vos modèles définis précédemment.

Il va falloir utiliser l'application **admin** qui est dans les contributions officielles de Django, ce qui consiste à déclarer dans les settings au niveau des **INSTALLED_APPS** (en plus des applications existantes)

```
'django.contrib.admin',
```

et de synchroniser une nouvelle fois la base de données

```
$ python manage.py syncdb
```

puis de rajouter les lignes suivantes à vos URL, dans un premier temps

```
from django.contrib import admin
admin.autodiscover()
```

et enfin cette ligne à ajouter sous celles existantes dans **urlpatterns** :

```
(r'^admin/(.*)', admin.site.root),
```

En vous rendant sur http://127.0.0.1:8000/admin/, vous devriez pouvoir vous identifier avec le compte créé lors de la création de la base de données précédemment. Si vous avez omis de le faire, il est toujours possible de créer ce super utilisateur en passant par le shell de Django :

```
$ python manage.py shell
>>> from django.contrib.auth.models import User
>>> User.objects.create_superuser(username="david", email="david@
larlet.fr", password="django")
```

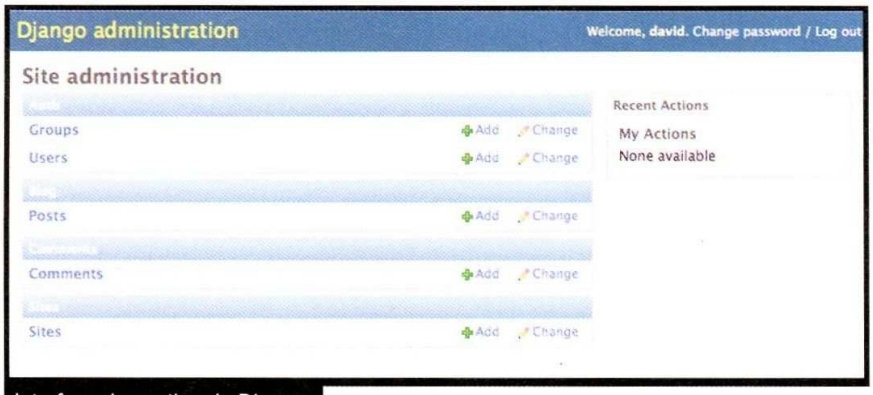
Le mot de passe va automatiquement être converti en une chaîne de caractères (*hash*) qui ne permettra pas de connaître le mot de passe en question, même pour un utilisateur ayant accès à la base de données. C'est une bonne pratique de sécurité. Notez que vous pouvez également utiliser un préfixe différent de **admin/** pour les URL si vous souhaitez cacher l'interface d'identification aux connaisseurs afin d'éviter les attaques par force brute.

Parfait, vous pouvez naviguer dans l'administration. Néanmoins, vous remarquerez que les objets de type **Post** qui constituent votre blog ne sont pas (encore !) accessibles. Il faut pour cela les déclarer dans un fichier **blog/admin.py** avec la fonction **admin.site.register** (encore une fois, c'est conventionnel, vous pouvez le faire directement dans les modèles si vous préférez) :

```
from django.contrib import admin
from projet_hors_serie.blog.models import Post

admin.site.register(Post)
```

Notez qu'il est nécessaire de redémarrer manuellement le serveur de développement lors de ce type de modification. Et voilà, vous êtes libre de créer, d'éditer et de supprimer vos billets grâce à cette interface.



Interface de gestion de Django

Initialement, cette interface était uniquement dédiée aux administrateurs (de confiance, par définition). Depuis la version 1.0, celle-ci a été réécrite et permet d'avoir une plus grande flexibilité permettant l'accès aux ressources en fonction des permissions par exemple. Il est donc envisageable, à moindres coûts, d'utiliser cette interface pour des utilisateurs à condition de bien gérer les permissions au niveau de la définition des classes définissant votre administration.

Plusieurs interfaces d'administration peuvent être générées pour un même site si vous souhaitez compartimenter les accès administrateur, modérateur, etc.

Notez que l'interface d'administration peut être utilisée pour administrer une base de données utilisée par un autre service. Vous pouvez ainsi avoir une interface d'administration cohérente entre votre site internet en Django, votre forum en Java et votre blog en PHP par exemple. La commande `inspectdb` permet de générer automatiquement les modèles à partir d'une base existante.

5.3 Zope 3

Le principe de Zope 2 était de faire du développement « via le web » en laissant la possibilité de créer un site ou une application complète depuis un navigateur web, grâce à une interface de gestion appelée « ZMI ». Celle-ci permettait d'ajouter du contenu, des modules additionnels, créer des gabarits de page et même des scripts Python. Paradoxalement, ce modèle de développement a fait le succès de Zope 2 au début des années 2000, mais a été progressivement abandonné à cause des nombreux problèmes qu'il engendre : entre autres, difficulté d'utiliser un outil de contrôle de version, mélange entre le code et les données, version restreinte de Python.

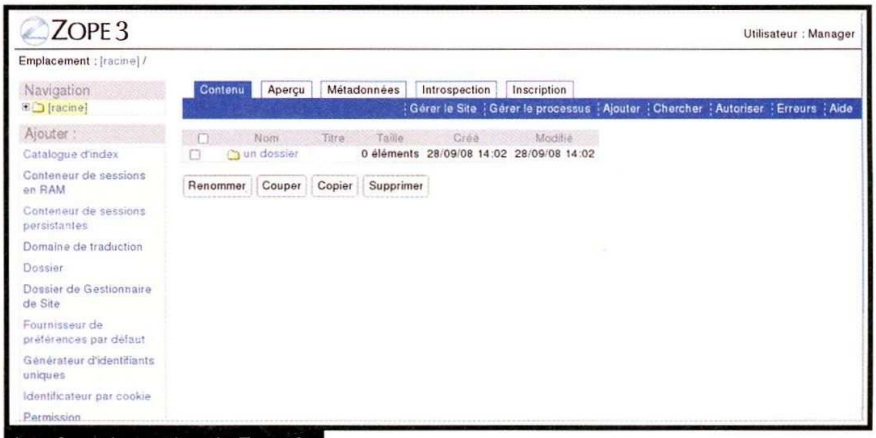
Le modèle de Zope 3 est beaucoup plus classique et l'interface de gestion, bien que toujours présente dans une version plus moderne et plus ergonomique, n'est plus du tout utilisée pour le développement. Elle peut être vue comme une sorte de « phpMyAdmin » de Zope et de la ZODB. En voici une capture :

Elle permet notamment de :

- naviguer dans l'arborescence des objets ;
- ajouter, supprimer ou modifier du contenu ;
- inspecter les objets Python stockés dans la base :
- leurs méthodes et attributs,
- leurs métadonnées,
- les interfaces qu'ils fournissent,
- les vues qui leur sont associées ;
- inscrire des objets dans les registres locaux ;
- naviguer dans le code source de Zope (et potentiellement de votre application) en inspectant toutes les classes et leur documentation ;
- effectuer des tâches d'administration comme le compactage de la base, l'arrêt du serveur.

Cette ZMI porte le nom de « *skin* Rotterdam », car elle a été créée en 2005 lors d'un sprint dans cette ville. Elle n'est rien d'autre qu'un exemple d'application Zope 3. Grâce à l'architecture de composants, elle peut être modifiée ou étendue à volonté. Vous en avez eu un aperçu dans l'article sur les contrôleurs. De cette façon, il est tout à fait possible de l'utiliser comme un outil de gestion de contenu prêt à l'emploi : elle représente quasiment un CMS complet à elle toute seule ! Avec un bémol : son caractère trop générique et trop près du framework la rendent impropre à une utilisation par un non-initié : en bref, vous pourrez l'utiliser vous-même, mais ne demandez pas à vos clients de l'utiliser pour gérer leur site. Une autre skin nommée « Minimal » existe et peut éventuellement servir de base à un site ou une application. Enfin, une alternative intéressante à la ZMI a vu progressivement le jour sous le nom de « ZAM » (*Zope Application Management*). ZAM est une sorte de ZMI simplifiée, encore plus modulaire et plus facilement personnalisable, qui utilise une extension de la Component Architecture nommée `z3c.baseregistry`.

Auteurs : Christophe Combelles, David Larlet
Zope3, Pylons, introductions : Christophe Combelles
Django : David Larlet
Conseils et relectures : Gael Pasgrimaud, Philippe Biondi



Interface de gestion de Zope 3

KDE RÉPOND-IL ENFIN À VOS ATTENTES ?

LINUX PRATIQUE 52

AU SOMMAIRE...

ENCORE DISPONIBLE CHEZ VOTRE MARCHAND DE JOURNAUX JUSQU'AU 30 AVRIL 2009

ANNEXE : Ressources et documentation

En guise de conclusion, cette dernière rubrique vous permettra de vous orienter dans vos recherches. Une grosse partie de la documentation se trouve sur le web, mais les vrais livres en papier peuvent être aussi très utiles. N'hésitez pas à rejoindre les canaux IRC et les listes de diffusion, car c'est là que se situe le cœur de la communauté, à même de vous apporter une aide précieuse. Chaque framework web possède ses propres moyens de communication, mais vous pouvez également rejoindre ceux sur Python, comme la liste Python de l'AFUL, les forums et les listes de l'AFPY, ainsi que les canaux #python-fr et #afpy sur Freenode. Enfin, l'AFPY organise régulièrement des rencontres Python (apéros ou sprints) qui sont souvent consacrées au web. Quel que soit votre niveau en Python, vous pourrez autant apprendre qu'apporter votre aide, notamment pour faire évoluer le site web de l'AFPY qui est un vrai melting-pot de plusieurs technologies, assemblées grâce à WSGI, où les trois frameworks présentés ici sont utilisés.

Pour terminer, il faut mentionner qu'un sujet primordial n'a pas été traité dans cette série d'articles par manque de place : les tests unitaires et fonctionnels. Chaque framework dispose de ses propres outils pour écrire et lancer des tests : Pylons utilise Nose, Django possède son propre outil intégré dans sa commande `manage.py` et Zope 3 utilise son paquet `zope.testing`. Tous trois font usage des outils natifs de Python, unittest et doctest, et encouragent à respecter une pratique courante : le développement dirigé par les tests et la documentation. On écrit d'abord les tests correspondant à ce qu'on veut obtenir, sous la forme de tests unitaires ou de doctests (de la documentation testable ou des tests documentés). L'implémentation vient ensuite. C'est une pratique qui semble un peu contraignante au premier abord et qui donne l'impression de ralentir les projets, mais qui, en réalité, diminue les temps globaux de développement et de débogage par un facteur 2 ou 3 !

Pylons

- Canal IRC sur Freenode.net : #pylons
- Liste de diffusion anglophone : **http://groups.google.com/group/pylons-discuss**

Vous pouvez également rejoindre le canal IRC de l'AFPY : #afpy sur Freenode. Certains composants tiers comme SQLAlchemy disposent de leur propre canal IRC sur Freenode.

- Pylons : **http://docs.pylonshq.com/**
- SQLAlchemy : **http://www.sqlalchemy.org/docs/05/index.html**
- repoze.who : **http://static.repoze.org/whodocs**
- repoze.what : **http://static.repoze.org/whatdocs/**
- Routes : **http://routes.groovie.org/manual.html**
- FormAlchemy : **http://docs.formalchemy.org**
- Mako : **http://www.makotemplates.org/docs/**

Un livre en anglais sur Pylons, le PylonsBook, est disponible en license GFDL :

- en version papier aux éditions Apress : *The definitive guide to Pylons*
- en ligne à l'adresse **http://pylonsbook.com**

Django

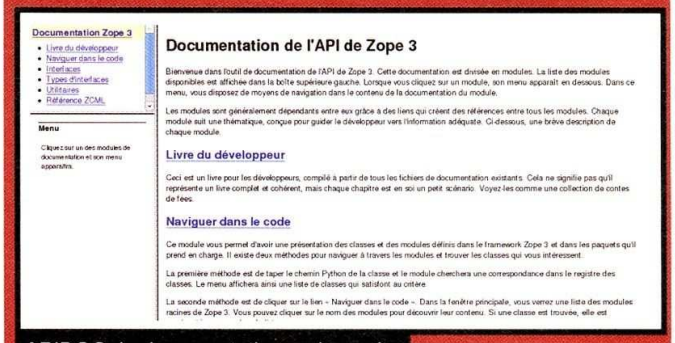
Il est à noter que de très nombreuses applications génériques que vous pourrez retrouver sur Google Code, GitHub ou BitBucket existent déjà.

- Documentation officielle : **http://docs.djangoproject.com**
- Documentation traduite : **http://www.django-fr.org/documentation/**
- Djangobook : **http://www.djangobook.com**
- Djangobook traduit : **http://djangobook.zindep.com**
- Blog officiel : **http://www.djangoproject.com/weblog/**
- Blogs parlant de Django : **http://www.djangoproject.com/community/**
- Actualité Django francophone : **http://www.biologeek.com/django/**
- Sites utilisant Django : **http://djangosites.org**
- Personnes utilisant Django : **http://djangopeople.net**
- Snippets dédié à Django : **http://djangosnippets.org**
- Liste de diffusion : **http://groups.google.com/group/django-users/**
- Liste de diffusion francophone : **http://lists.afpy.org/mailman/listinfo/django**

Zope 3

- Liste de diffusion francophone : voir http://lists.afpy.org
- Canal IRC francophone sur Freenode.net : #zope3-fr
- Liste anglophone : Zope3-users sur http://lists.zope.org
- Canal IRC anglophone sur Freenode.net : #zope3-dev

La documentation intégrée de l'API est disponible après démarrage de Zope 3 en cliquant sur « Aide » dans l'interface de gestion (ZMI). Vous pouvez aussi y accéder directement à **http://localhost:8080/++apidoc++/**.



APIDOC, la documentation embarquée

Le livre (en anglais) de Philip Von Weitershausen, aux éditions Springer, *Web component development with Zope 3*, est extrêmement bien écrit et vous permettra de plonger dans Zope 3 en un temps record. Un autre livre a été écrit il y a quelques années par Stephan Richter et est disponible en ligne. Il n'est plus à jour par rapport à la version actuelle, mais est quand même digne d'intérêt.

- Page d'annonce Zope 3.4.0 : **http://download.zope.org/zope3.4/**
- Site web de Zope : **http://zope.org**
- Nouveau site en cours de réalisation : **http://new.zope.org**
- Version statique de l'APIDOC : **http://apidoc.zope.org**
- Zope Component Architecture : **http://www.muthukadan.net/docs/zca.html**
- z3c tutorial : **http://www.carduner.net/docs/z3c-tutorial/**

Abonnez-vous !

Vous lisez d'autres magazines des Éditions Diamond ?
Des offres de couplage sont disponibles au verso.

Économisez Plus de 20%*

* Sur le prix de vente unitaire France Métropolitaine

11 Numéros de GNU/Linux Magazine pour le prix de 9*

* Gain pour un abonnement France Métropolitaine, par rapport au prix unitaire France Métropolitaine

11 Numéros de GNU/Linux Magazine à 55 €
(Offre France Métro)

Soit votre GNU/Linux Magazine à 5,00 €
(Tarif au numéro dans le cadre d'un abonnement France Métro)

Les 3 bonnes raisons de vous abonner !

- Ne manquez plus aucun numéro.
- Recevez GNU/Linux Magazine chaque mois chez vous ou dans votre entreprise.
- Économisez 16,50 €/an ! (soit plus de 2 magazines offerts !)

Pour les tarifs hors France Métropolitaine, consultez notre site : www.ed-diamond.com.

Bon d'abonnement à découper

Tournez SVP pour découvrir toutes les offres d'abonnement >>

GNU LINUX MAGAZINE / FRANCE

Édité par Les Éditions Diamond
Tél. : + 33 (0) 3 88 58 02 08
Fax : + 33 (0) 3 88 58 02 09

Voici mes coordonnées postales :
Nom :
Prénom :
Adresse :
Code Postal :
Ville :

Vos remarques :

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **www.ed-diamond.com/cgv** et reconnais que ces conditions de vente me sont opposables.

Offres d'abonnement

(Nos tarifs s'entendent TTC et en euros)

		F	D	T	E1	E2	EUC	A	RM
		France Métro	DOM	TOM	Europe 1	Europe 2	Etats-unis Canada	Afrique	Reste du Monde
1	Abonnement Linux Magazine	55 €	59 €	67 €	69 €	66 €	70 €	68 €	77 €
2	Linux Magazine + Hors-série	83 €	89 €	101 €	104 €	100 €	105 €	103 €	116 €
3	Linux Magazine + MISC	84 €	90 €	102 €	105 €	101 €	107 €	104 €	117 €
4	Linux Magazine + Linux Pratique	78 €	85€	96 €	99 €	95 €	101 €	98 €	111 €
5	Linux Magazine + Hors-série + Linux Pratique	110 €	119 €	134 €	138€	133 €	140 €	137 €	154 €
6	Linux Magazine + Hors-série + MISC	116 €	124 €	140 €	144 €	139 €	146 €	143 €	160 €
7	Linux Magazine + Hors-série + MISC + Linux Pratique	143 €	154 €	173 €	178 €	172 €	181 €	177 €	198 €
8	Linux Pratique Essentiel + Linux Pratique	57 €	62 €	69 €	71 €	69 €	73 €	71 €	79 €

- Europe 1 : Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède
- Europe 2 : Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Islande, Suisse, Irlande
- Zone Reste du Monde : Autre Amérique, Asie, Océanie
- Zone Afrique : Europe de l'Est, Proche et Moyen-Orient

Toutes les offres d'abonnement : en exemple les tarifs ci-dessous correspondant à la zone France Métro (F)

(Vous pouvez également vous abonner sur : www.ed-diamond.com)

Linux Magazine (11 n°s)

1 par ABO : 55€ — Economie : 16,50 €

en kiosque : 71,50€

Linux Magazine (11 n°s) + Linux Magazine hors-série (6 n°s)

en kiosque : 110,50€

2 par ABO : 83€ — Economie : 27,50 €

Linux Magazine (11 n°s) + Misc (6 n°s)

en kiosque : 119,50€

3 par ABO : 84€ — Economie : 35,50 €

Linux Magazine (11 n°s) + Linux Pratique (6 n°s)

en kiosque : 107,20€

4 par ABO : 78€ — Economie : 29,20 €

Linux Pratique Essentiel (6 n°s) + Linux Pratique (6 n°s)

en kiosque : 74,70€

par ABO : 57€ — Economie : 17,70 €

8

5 Linux Magazine (11 n°s) + Linux Magazine hors-série (6 n°s) + Linux Pratique (6 n°s)

par ABO : 110€ — Economie : 36,20 €

en kiosque : 146,20€

6 Linux Magazine (11 n°s) + Linux Magazine hors-série (6 n°s) + Misc (6 n°s)

par ABO : 116€ — Economie : 42,50 €

en kiosque : 158,50€

Linux Magazine (11 n°s) + Linux Magazine hors-série (6 n°s) + Misc (6 n°s) + Linux Pratique (6 n°s)

en kiosque : 194,20€

7 par ABO : 143€ — Economie : 51,20 €

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous :

Je fais mon choix de la 1ère offre :

Je sélectionne le N° (1 à 8) de l'offre choisie :	
Je sélectionne ma zone géographique (F à RM) :	
J'indique la somme due : (Total 1)	€

Je fais mon choix de la 2ème offre :

Je sélectionne le N° (1 à 8) de l'offre choisie :	
Je sélectionne ma zone géographique (F à RM) :	
J'indique la somme due : (Total 2)	€
Montant Total à régler (Total 1 + Total 2)	€

Exemple : je souhaite m'abonner à l'offre Linux Magazine + Hors-série + MISC (offre 6) et je vis en Belgique (E1), ma référence est donc 6E1 et le montant de l'abonnement est de 144 euros.

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre de Diamond Editions
- ❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

LINUX MAGAZINE / FRANCE

Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex

Avez-vous l'âme du collectionneur ?

Boostez votre collection !

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 GNU/Linux Magazine pour 15€ et les « Power packs x10 », soit 10 GNU/Linux Magazine pour 25€ à choisir dans la liste ci-dessous :

Les 4 façons de commander !

Par courrier
En nous renvoyant ce bon de commande.

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 x5 et x10

N°06	GNOME - The Gimp
N°07	Dopez Linux
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°22	Le multi-hreading : Une manière moderne de programmer le Multitâche
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°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	Steganographie ou l'art de la dissimulation de données
N°44	Comprenez NetBios pour Maîtriser l'interopérabilité windows GNU Linux
N°46	Debian : Utilisez Samba avec le support ACL
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°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, Spécifications, protocoles et configuration
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 solution libre pour systèmes embarqués
N°83	Greylist Eliminez le SPAM à la racine
N° 84	Déploiement de hotspots Wifi sécurisés
N° 85	Firewall : Netfilter & NuFW
N° 86	Serveur SMTP: Routage des mails avec Postfix
N° 87	Le point sur Mono .NET Java et les Brevets
N° 88	Sécurité: Smartcards & Tokens
N° 89	Utilisation avancée de XEN
N° 91	Ajax Avancé
N° 92	Paravirtualisation XEN & SLO Répartition de charge
N° 93	Développez vos extensions Firefox
N° 94	PABX Vidéo avec Asterisk
N° 95	Administration et configuration centralisées avec Cfengine
N° 96	Géolocalisation de photos numériques
N° 97	Haute-disponibilité & équilibrage de charge
N° 98	Supervision avec NAGIOS
N° 99	Java & JNI Tirez le meilleur de Java
N°100	Le serveur PARFAIT
N°101	Chiffrez vos disques
N°102	LTSP 5 Clients légers
N°103	PostgreSQL 8.3

Numéros GNU/Linux Magazine épuisés
N°01, N°02, N°03, N°04, N°05, N°08, N°20, N°21, N°23, N°31, N°33, N°42, N°43, N°45, N°47, N°54 et N°90

Bon de commande power packs

à remplir (ou photocopie) et à retourner aux Éditions Diamond – GNU/Linux Magazine – BP 20142 – 67603 sélestat Cedex

	OUI, je désire acquérir un power pack X5	1er 1PP* X5	2éme 2PP* X5	3éme 3PP* X5
Cochez ici ▲ POWER PACKS X5	1, GNU/Linux Magazine N°			
	2, GNU/Linux Magazine N°			
	3, GNU/Linux Magazine N°			
	4, GNU/Linux Magazine N°			
	5, GNU/Linux Magazine N°			
	Total par série de POWER PACKS X5 :	15 €	30 €	45 €
	OUI, je désire acquérir un power pack X10	**1er 1PP* X10**	**2éme 2PP* X10**	**3éme 3PP* X10**
Cochez ici ▲ POWER PACKS X10	1, GNU/Linux Magazine N°			
	2, GNU/Linux Magazine N°			
	3, GNU/Linux Magazine N°			
	4, GNU/Linux Magazine N°			
	5, GNU/Linux Magazine N°			
	6, GNU/Linux Magazine N°			
	7, GNU/Linux Magazine N°			
	8, GNU/Linux Magazine N°			
	9, GNU/Linux Magazine N°			
	10, GNU/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 :		

Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Je choisis de régler par :

❑ Chèque bancaire ou postal à l'ordre de Diamond Editions

❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Complétez votre collection des anciens numéros de...

Les 4 façons de commander !

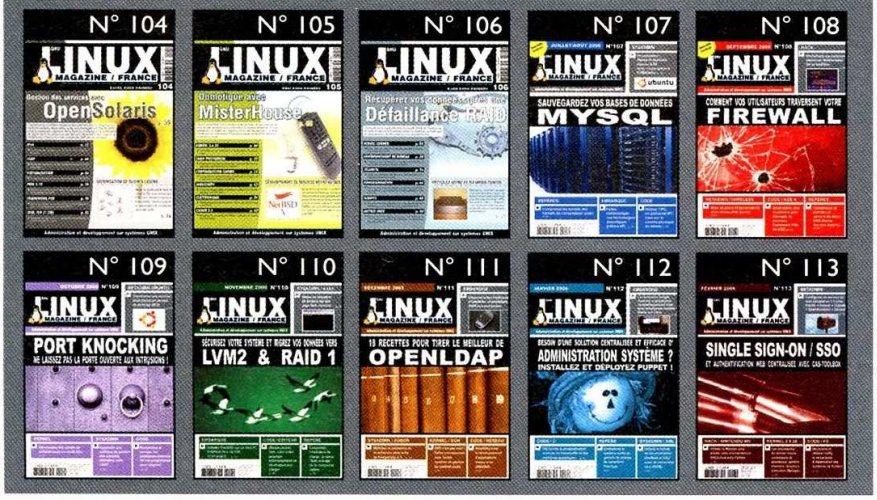
Par courrier
En nous renvoyant ce bon de commande.

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.

GNU/LINUX MAGAZINE



Bon de commande GNU/Linux Magazine

Réf.	Désignation	Prix / N°s
LM104	**Linux Magazine 104** Gestion des services avec OpenSolaris	6,50 €
LM105	**Linux Magazine 105** Domotique avec MisterHouse	6,50 €
LM106	**Linux Magazine 106** Récupérez vos données après une Défaillance RAID	6,50 €
LM107	**Linux Magazine 107** Sauvegardez vos bases de données MySql	6,50 €
LM108	**Linux Magazine 108** Comment vos utilisateurs traversent votre Firewall	6,50 €
LM109	**Linux Magazine 109** Port Knocking - Ne laissez pas la porte ouverte aux intrusions	6,50 €
LM110	**Linux Magazine 110** Sécurisez votre système en le migrant vers LVM2 & RAID 1	6,50 €
LM111	**Linux Magazine 111** 18 Recettes pour tirer le meilleur de OPENLDAP	6,50 €
LM112	**Linux Magazine 112** Besoin d'une solution centralisée et efficace d'ADMINISTRATION SYSTÈME ? Installez et déployez Puppet !	6,50 €
LM113	**Linux Magazine 113** SINGLE SIGN-ON / SSO et authentification web centralisée avec CASTOOLBOX	6,50 €

GNU/LINUX MAGAZINE HORS-SÉRIE

Bon de commande GNU/Linux Magazine Hors-série

Réf.	Désignation	Prix / N°s
LMHS24	**LM HS 24** Linux Embarqué	6,40 €
LMHS25	**LM HS 25** Linux Embarqué 2	6,40 €
LMHS26	**LM HS 26** Spécial The GIMP	6,40 €
LMHS27	**LM HS 27** Électronique et Linux	6,40 €
LMHS28	**LM HS 28** Administration système avec Debian	6,40 €
LMHS29	**LM HS 29** BSD Acte 1	6,40 €
LMHS30	**LM HS 30** BSD Acte 2	6,40 €
LMHS31	**LM HS 31** Spécial The GIMP	6,40 €
LMHS32	**LM HS 32** VIRUS Unix, Gnu/Linux & Mac OS X	6,40 €
LMHS33	**LM HS 33** Ruby & Ruby on Rails	6,40 €
LMHS34	**LM HS 34** Dominez votre système et allez au-delà de l'interface graphique...	6,50 €
LMHS35	**LM HS 35** Serveur Web, Installez, configurez et optimisez votre serveur HTTP !	6,50 €
LMHS36	**LM HS 36** Serveur Mail, Installez, configurez et optimisez votre serveur SMTP !	6,50 €
LMHS37	**LM HS 37** Serveur Dédié, Supervision, VoIP, VPN, Syslog...	6,50 €
LMHS38	**LM HS 38** Électronique Embarqué & Domotique	6,50 €
LMHS39	**LM HS 39** Cartes à puce administration et utilisation	6,50 €

Bon de commande

à remplir (ou photocopie) et à retourner aux Éditions Diamond – GNU/Linux Magazine – BP 20142 – 67603 sélestat Cedex

	Référence	Prix / N°s	Qté	Total
EXEMPLE :	LM111	6,50 €	1	6,50 €
		TOTAL :		
		FRAIS DE PORT FRANCE MÉTRO :		+3,81 €
		FRAIS DE PORT ÉTRANGER :		+5,34 €
		TOTAL :		

Voici mes coordonnées postales :
Nom :
Prénom :
Adresse :
Code Postal :
Ville :

Je choisis de régler par :

❑ Chèque bancaire ou postal à l'ordre de Diamond Editions

❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

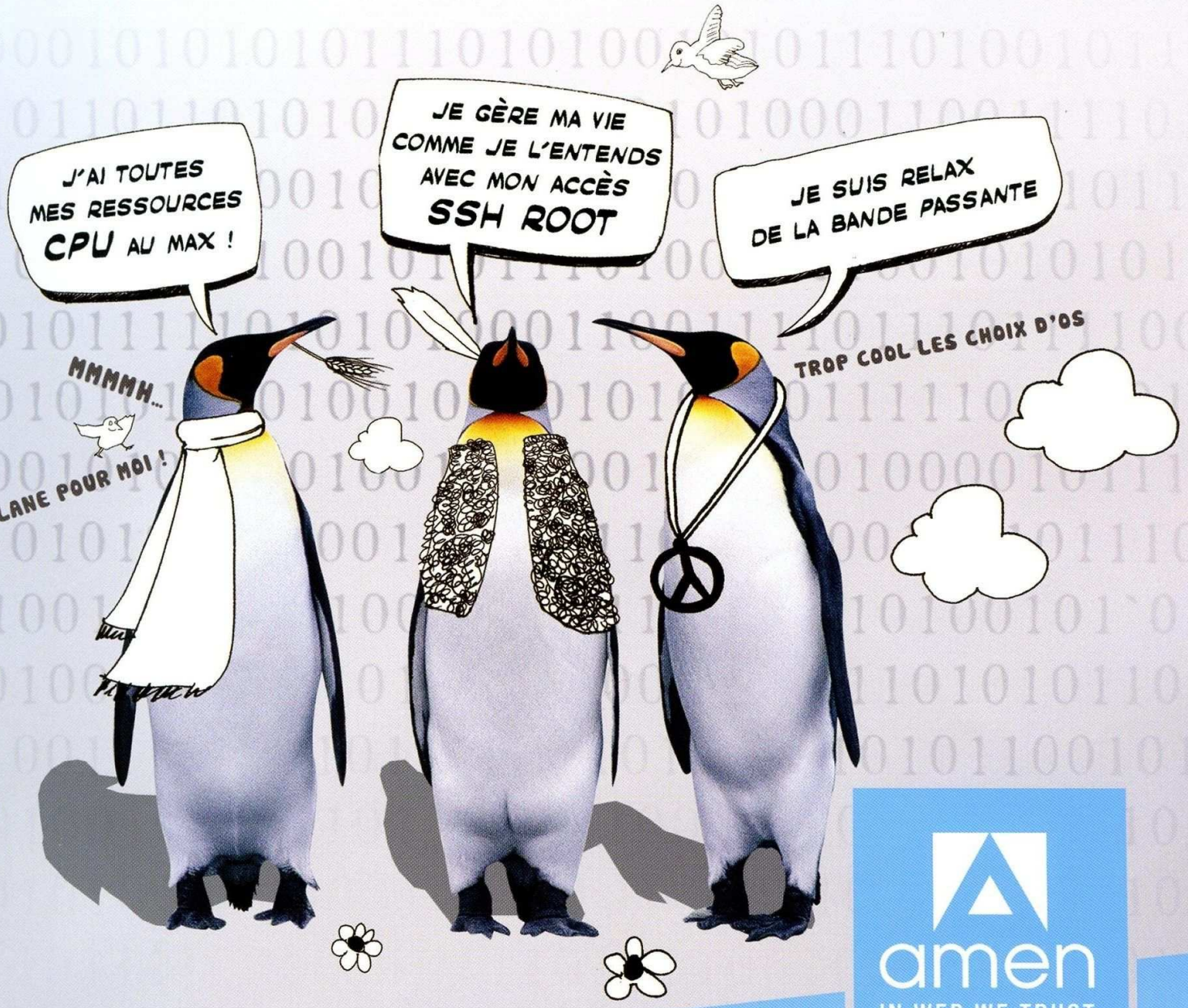
Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Retrouvez les sommaires et commandez tous nos magazines sur notre site :
http://www.ed-diamond.com

NOUVEAU VDS+ d'AMEN :
le bonheur est dans le serveur !



amen
IN WEB WE TRUST

À PARTIR DE
5 € HT /MOIS
soit 5,98 € TTC/MOIS*

SERVEURS PRIVÉS AMEN : BÉNÉFICIEZ DE RESSOURCES GARANTIES QUI VOUS SONT PROPRES (PROCESSEUR, MÉMOIRE, DISQUE DUR...) TOUT EN PROFITANT D'UNE PLATEFORME INFOGÉRÉE 24H/24 - 7J/7.

- Hébergement multi-sites/multi-domaines
- Interface d'administration : Plesk 8.6
- Systèmes d'exploitation : Fedora Core 8, Suse 10.3, Debian 4.0, Ubuntu 8.04 ou CentOS 5
- Part CPU minimum : de 1 à 6
- Mémoire garantie : de 256 Mo à 1 Go
- Espace disque : de 5 Go à 30 Go
- Bases de données : illimitées
- 1 adresse IP fixe
- Accès Root

PARTENAIRE INDUSTRIEL
entrepreneurs, faites le choix de l'économie numérique
www.econumerique.pme.gouv.fr

Pour plus de renseignements : 0892 55 66 77 (0.34 €/mn) ou www.amen.fr
NOMS DE DOMAINE - EMAIL - HÉBERGEMENT - CRÉATION DE SITE - E-COMMERCE - RÉFÉRENCEMENT